推酷

编程狂人

Programming Madman

NO.39

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:http://www.tuicool.com/mags/53fb42a3d91b1406170486e4

欢迎下载推酷客户端体验更多阅读乐趣

目录

开源的 PHP 轻量级框架 iphp

作者：ideawu

对于 PHP Web 开发来说, 框架很重要, 但其实框架又不重要. 说重要, 是因为确实需要一个框架来建立一套规范, 对文件组织, 类和方法的编写, 数据库操作等进行引导. 说不重要, 是因为对于 PHP 开发, 没有哪一个框架最必须的, 不是框架不给力, 而是 PHP 的世界从来就没有呼唤这样的框架的出现.

在 PHP 框架既重要又不重要的事实面前, 所有的 PHP 框架必须是轻量级的, 如果一个 PHP 框架非常重, 那么它肯定没有前途, 不是流行不起来, 就是流行很短暂便沉寂.

我建议每一个团队都开发自己的 PHP 框架, 简单就好. 最好是那种 3 天就开发出来的框架, 而不是那种憋了一年半载才勉强做出来奇怪且复杂的框架. 这种框架必须是透明的, 即使是新手也能一眼看穿框架的核心. 如果做不到的话, 我建议还是换人(没错, 这种情况下人应该被换掉).

对于 PHP 框架, 我有一些要求:

1. 不能做太多事

PHP 框架不要总想做所有事, 缓存系统不需要框架来做, Session 管理也不需要, 存储层封装不要太过度以至搞出各种恶心的 ORM, ActiveRecord 之类的无用功能. 这些功能和模块, 应该独立于框架, 采用成熟的技术.

2. 不要”创造”所谓的模板语言

PHP 语言本身就是模板语言, PHP 做模板语言对于 PHP Web 来说是最完美的, 可维护性和培训成本最佳的语言, 只需要再多说一两句话规范即可: 仅使用 echo 及允许的帮助 echo 的函数, 和 if/for/while. 我十年前不认同 smarty 这类模板工具的意义, 十年后也不认可这类毫无意义的寄生于 PHP 的工具.

3. 使用 PHP 框架的最佳状态是忘掉框架

框架要足够简便, 功能恰到好处, 没有不必要的限制, 这样在使用的过程中才能让人忘掉框架的存在, 以便能将精力放在业务本身. 当需要开发一个功能时, 程序员想的不应该是”框架能不能做”, 而是”我能不能做”.

4. 最后

我自己也开发了一个轻量级的 PHP 框架, 命名为 iphp. iphp 非常简便和轻量, 全部有效代码不过一千行. iphp 只解决 Web 开发中最重要的问题: 代码组织, URL路由和URL生成.

这个框架用来开发了 SSDB 数据库的图形化界面管理工具 phpssdbadmin(开源项目). 如果你去看 phpssdbadmin 你就会发现, iphp 做到了前面的要求, phpssdbadmin 的用处几乎都没有注意到 PHP 框架的存在.

相关链接:

iphp框架入门(https://github.com/ideawu/iphp#%E9%A1%B9%E7%9B%AE%E6%96%87%E4%BB%B6%E7%BB%84%E7%BB%87%E7%9B%AE%E5%BD%95%E7%BB%93%E6%9E%84)

iphp框架如何生成URL(https://github.com/ideawu/iphp/wiki/URL-%E7%9A%84%E7%94%9F%E6%88%90)

原文链接:http://www.ideawu.net/blog/archives/828.html

高密度Java应用部署的一些实践

作者：李三红

传统的Java应用部署模式，一般遵循“硬件->操作系统->JVM-> Java应用”这种自底向上的部署结构，其中JEE应用 可以细化为“硬件->操作系统->JVM->JEE容器-> JEE应用”的部署结构。这种部署结构往往比较重，操作系统、JVM 和JEE容器造成的overhead很高，而很多时候一个Java应用并不需要跑满整个硬件的资源，导致这种模式的资源利用率是比较低的。

而另一方面，硬件虚拟化技术逐渐成熟：VMware Hypervisor、Xen 、KVM、Power LPAR等技术能够帮助我们在同一个硬件上部署多个操作系统实例，而时下流行的OS Container技术如LXC、Docker等，则是把操作系统虚拟化为多个实例，实现更轻量级的虚拟化。无论哪个层面的虚拟化，其目的都是对资源利用 率更加高效的追求，从而成为如今构建云计算平台底层架构的基础技术。

Java应用也可以通过同样的思路来实现高密度的部署。JVM虚拟化是比OS虚拟化更高一层的做法，可以更大程度的提高资源利用率，降低平均应用的部署成本。本文将介绍Multi-tenant JVM这一方案实现高密度Java应用部署的一些特点和思路。

背景介绍

早在2004年，Sun公司就提出过Java应用虚拟化这方面的想法。当时Grzegorz Czajkowski领导了一个叫做巴塞罗那的研究项目，该项目基于Java HotSpot虚拟1.5版本开发了Multi-Tasking Virtual Machine（MVM）。MVM的目的旨在提高Java程序的启动速度，节省内存开销。不过自从Sun被甲骨文收购后，我们没有听到关于该项目的任何新 的进展。

尽管我们没有看到MVM成功产品化，不过它却留下两个JSR规范：JSR121和JSR284。对于JSR284，目前在java.net上有一个实现它的孵化项目。

从2009年开始，我所在的IBM Java团队开始研究Java应用的SaaS化方案，即让一个应用实例服务于多个租户。为了保证多个租户在使用同一个应用实例时候数据的隔离，该方案在应 用这个层面做了一些Bytecode Instrument（BCI）的工作，主要通过改写getstatic/putstatic使每个租户有独立的类的静态数据拷贝而没有相互影响。但是， 该方案在Byteco-de层面更改带来的额外性能开销， 以及Java Reflection等访问带来的安全性/正确性的问题。 而且，除了数据上的隔离，也需要针对关键性的资源譬如CPU、Heap、IO等资源的使用进行管理，于是该方案下沉到了JVM层面，形成现在的多租户JVM（Multi-tenant JVM）方案。

Multi-tenant JVM是JVM层面的虚拟化，其思路是把多个Java应用部署在同一个JVM上，让这些应用共享底层的GC、JIT、Java运行时库等基础组件。除了IBM的团队之外，爱尔兰的Waratek公司也 实现了多租户的JVM。和IBM Multi-tenant JVM类似，Waratek允许多个应用运行在同一个CloudVM上，每一个应用运行在一个叫Java Virtual Contain-er（JVC）的容器里。从现有公开的资料开看，IBM Multi-tenant JVM是基于Java 7的，而Waratek是基于Java 6的，两者支持的CPU架构和平台也有所不同。

此外，JEE方面在两年前也有讨论计划增加对PaaS和多租户的支持，这项提议旨在定义PaaS环境下如何使得JEE应用支持多租户，保证不同租户在使用这些应用时相互隔离，以及资源方面的管理（如JMS资源），不过该项提议已经推迟到JEE 8。

除了提升部署密度之外，多租户的另一项好处在于应用启动的加速。快速的程序启动受益于不同的应用共享同一个JVM，我们称之为javad。Java核心的类库在javad运行后，不再需要被重新装载和定义。你也许可以用Nailgun来加速你的启动时间，但Nailgun的问题是没有安全的数据隔 离，这包括类的静态数据以及Java属性值，而且Nailgun在易用性等方面也不如Multi-tenent JVM。

多租户JVM的实现思路

跟传统JVM相比，多租户JVM的主要工作围绕隔离而进行，其针对JVM/JDK的改动主要实现三个方面的目标：

1. 租户之间的数据隔离
2. Java类库支持多租户语境
3. 资源管理隔离

租户之间的数据隔离

让每个租户应用拥有独立的类静态数据拷贝，这个目标主要通过修改getstatic/putstatic字节码指令实现。下面是一个简单的例子：

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

每一个运行在Multi-tenant JVM上的程序都有不同System.out实例。就java.lnag.System内部实现来说，out是其类静态变量：

```
public final class System {


    // The standard input, output, and error streams.
    // Typically, these are connected to the shell which
    // ran the Java program.
    /**
     * Default input stream
     */
    public static final InputStream in = null;
```

```
    /**
     * Default output stream
     */
    public static final PrintStream out = null;
    /**
     * Default error output stream
     */
    public static final PrintStream err = null;
……
}
```

Multi-tenant JVM对于标准的JVM行为进行的更改如下：

- 每一个租户第一次使用java/lang/System时，都会触发它的初始化，也就是<clinit>。而一般的JVM，java/lang/System只会被初始化一次。
- <clinit>的执行，对于每一个静态成员变量存取，都被重新定向到了具体的租户存贮空间。比如对于out = null赋值，put-static执行时实际上会找到当前的租户，然后把值存到该租户的空间去，get-static有着类似的道理。

Java类库支持多租户语境

这部分主要通过改造类库实现，具体的功能包括：

- System.exit(code) 调用只会使当前租户退出，而不会令整个JVM退出。而租户申请的一些诸如File/Socket句柄之类系统资源，会随着租户的推出而被释放。
- 租户A不可能通过类似如下

```
ThreadGroup group = Thread.currentThread().getThreadGroup();
ThreadGroup parent = group.getParent();
```

枚举线程的办法获得租户B的线程。不同租户的线程分属于不同的线程组。

- Java属性值的隔离，比如同样的语句System.getProperty("name")对于不同的租户可能是不同的值。

资源管理隔离

这是Multi-tenant JVM很重要的功能。在Multi-tenant JVM上，Heap/CPU/Disk IO/Net IO这些资源的使用是受资源策略保护的，比如你可以去限制某个租户它的CPU最少可以使用20%，而在系统空闲时，最大可以用到100%。

Multi-tenant JVM通过Token Bucket来 对IO（Disk/Net）和CPU资源进行管理。对于IO而言，Multi-tenant JVM截获IO有关的OS API调用，使得IO发生之前受制于我们预先规定的资源策略。我们举个网络IO的例子，例如Java程序从Socket的读取操作，JDK内部的实现通过JNI实际上会对应到系统的API调用

*ssize_t recv(int sockfd, void *buf, size_t len, int flags);*

在recv调用发生之前，Multi-tenant JVM通过资源策略保证租户的IO使用带宽不会超过给它设定的限制。关于网络IO，我们这里有一个很好的演示：

用简单的-Xlimit:netIO=6M参数限制运行在Multi-tenant JVM上的Ftp Server带宽上限读写各为6Mib/s。

关于对CPU管理，Multi-tenant JVM实现的基本的思路是，把租户线程所花费的CPU时间量化为Tokens，运行时每一个租户线程都会被周期性检查是否其当前CPU时间的使用超过了给 它设定的限制。如果超过，当前线程会被挂起，直到满足限制为止。周期性检查的代码是由Multi-tenant JVM插入到租户线程里去的，对于用户程序而言完全是透明的。

Multi-tenant JVM对于Heap的管理建立在Balanced GC Policy基 础之上。同一般的Java程序类似，你可以使用-Xms/-Xmx为租户程序设定最大/最小的堆内存值。Balanced GC Policy基于Region对Heap进行管理，每个

租户程序根据-Xms/-Xmx的设定来为其分配Region，而租户对象的分配也必然只能发生在 它自己拥有的区域内。

多租户JVM的用法与限制

IBM发布的Java 7 R1默认支持多租户JVM，在命令行上添加-Xmt参数即可启用。由于多租户JVM对JVM 的变更，JNI Native Libraries、JVMTI以及GUI programs在多租户状态下的使用是受限制的。Multi-tenant JVM并未实现对JNI的隔离，所以不同的租户应用不能装载依赖同样的JNI Native Lib，所有发生在JNI Native Lib里的IO，不会受限于该租户资源消费策略。同样的情况适用于CPU以及Memory。

Multi-tenant JVM目前没有实现对JVMTI Agent的改造用以支持我们前面所描述的静态数据的隔离，这可能会对用户如果想调试Java核心类库代码（不是用户代码）造成困扰。

关于GUI，Multi-tenant JVM没有实现底层对于UI程序消息队列的隔离，所以不支持在同一个Multi-tenant JVM运行大于1个的GUI程序。

还有一点，不要在非Daemon线程里写 “暴力”的死循环代码，例如：

```
while(true)
{
try () {
    ....
} catch(Throwable t) {
{
}


}
```

最后需要注意的是，当开启IO资源控制时，尽量一次写出更多的字节，避免影响程序的IO性能。

总结

Multi-tenant JVM目前在应用启动时间和更小的内存占用开销方面已经被证实有效。根据目前的一些基准测试结果来看，对于简单的应用，相较于一般JVM，Multi- tenant JVM可以获得5~6倍的运行个数。后续计划发布的版本仍然会集中在提高启动时间、更小的内存开销这两个方面，也会陆续有一些性能的报告发布。

长远来看，Multi-tenant JVM会基于用户、IBM产品线以及技术社区等的反馈，做进一步的提高，以及解决一些目前所存在的局限性，比如对于JNI隔离的支持，JVMTi的多租户支持等等。

关于作者：李三红，IBM资深软件工程师，Multi-tenant JVM项目技术负责人，目前供职于IBM Java技术中心，从事多租户Java虚拟机相关的研发工作。九年多的Java开发经验，2008年加入IBM，参与基于OSGi框架的安全方面的开 发，2010年加入Java技术中心，参与IBM Java虚拟机 J9的开发。在Java技术领域拥有多项专利以及在developerWorks上发表十余篇文章。

原文链接：http://www.infoq.com/cn/articles/java-application-deployment-practise

用“逐步排除”的方法定位Java服务线上“系统性”故障

作者：李斯宁

一、摘要

由 于硬件问题、系统资源紧缺或者程序本身的BUG，Java服务在线上不可避免地会出现一些“系统性”故障，比如：服务性能明显下降、部分（或所有）接口超 时或卡死等。其中部分故障隐藏颇深，对运维和开发造成长期困扰。笔者根据自己的学习和实践，总结出一套行之有效的“逐步排除”的方法，来快速定位Java 服务线上“系统性”故障。

二、导言

Java 语言是广泛使用的语言，它具有跨平台的特性和易学易用的特点，很多服务端应用都采用Java语言开发。由于软件系统本身以及运行环境的复杂性，Java的 应用不可避免地会出现一些故障。尽管故障的表象通常比较明显（服务反应明显变慢、输出发生错误、发生崩溃等），但故障定位却并不一定容易。为什么呢？有如 下原因：

1. 程序打印的日志越详细，越容易定位到BUG，但是可能有些时候程序中没有打印相关内容到日志，或者日志级别没有设置到相应级别

2. 程序可能只对很特殊的输入条件发生故障，但输入条件难以推断和复现

3. 通常自己编写的程序出现的问题会比较容易定位，但应用经常是由多人协作编写，故障定位人员可能并不熟悉其他人员编写的程序

4. 应用通常会依赖很多第三方库，第三方库中隐藏着的BUG可能是始料未及的

5. 多数的开发人员学习的都是“如何编写业务功能”的技术资料，但对于“如何编写高效、可靠的程序”、“如何定位程序故障”却知之甚少。所以一

旦应用出现故障，他们并没有足够的技术背景知识来帮助他们完成故障定位。

尽管有些故障会很难定位，但笔者根据学习和实践总结出一套“逐步排除”的故障定位方法：**通过操作系统和Java虚拟机提供的监控和诊断工具，获取到系统资源和目标服务（出现故障的Java服务）内部的状态，并依据服务程序的特点，识别出哪些现象是正常的，哪些现象是异常的。而后通过排除正常的现象，和跟踪异常现象，就可以达到故障定位的目标。**

在正式介绍该方法之前，先申明一下这个方法使用的范围。

三、本方法适用的范围

本方法主要适用于Linux系统中Java服务线上“系统性”故障的定位，比如：服务性能明显下降、部分（或所有）接口超时或卡死。其它操作系统或其它语言的服务，也可以参考本文的思路。

不适用本方法的情况：对于“功能性”故障，例如运算结果不对、逻辑分支走错等，不建议使用本方法。对待这些情况比较恰当的方法是在测试环境中重现，并使用Java虚拟机提供的“远程调试”功能进行动态跟踪调试。

前面说过，本方法基于“异常现象”的识别来定位故障。那系统中可能有哪些异常现象呢?

四、有哪些异常现象

我们可以将异常现象分成两类：系统资源的异常现象、“目标服务”内部的异常现象。目标服务，指的是出现故障的Java服务。

1. 系统资源的异常现象

一 个程序由于BUG或者配置不当，可能会占用过多的系统资源，导致系统资源匮乏。这时，系统中其它程序就会出现计算缓慢、超时、操作失败等“系统性”故障。 常见的系统资源异常现象有：CPU占用过高、物理内存富余量极少、磁盘I/O占用过高、发生换入换出过多、网络链接数过多。可以通过top、iostat、vmstat、netstat工具获取到相应情况。

2. 目标服务内部的异常现象

- Java堆满

Java堆是“Java虚拟机”从操作系统申请到的一大块内存，用于存放Java程序运行中创建的对象。当Java堆满或者较满的情况下，会触发 “Java虚拟机”的“垃圾收集”操作，将所有“不可达对象”（即程序逻辑不能引用到的对象）清理掉。有时，由于程序逻辑或者Java堆参数设置的问题， 会导致“可达对象”（即程序逻辑可以引用到的对象）占满了Java堆。这时，Java虚拟机就会无休止地做“垃圾回收”操作，使得整个Java程序会进入 卡死状态。我们可以使用jstat工具查看Java堆的占用率。

- 日志中的异常

目标服务可能会在日志中记录一些异常信息，例如超时、操作失败等信息，其中可能含有系统故障的关键信息。

- 疑难杂症

死锁、死循环、数据结构异常（过大或者被破坏）、集中等待外部服务回应等现象。这些异常现象通常采用jstack工具可以获取到非常有用的线索。

了解异常现象分类之后，我们来具体讲讲故障定位的步骤。

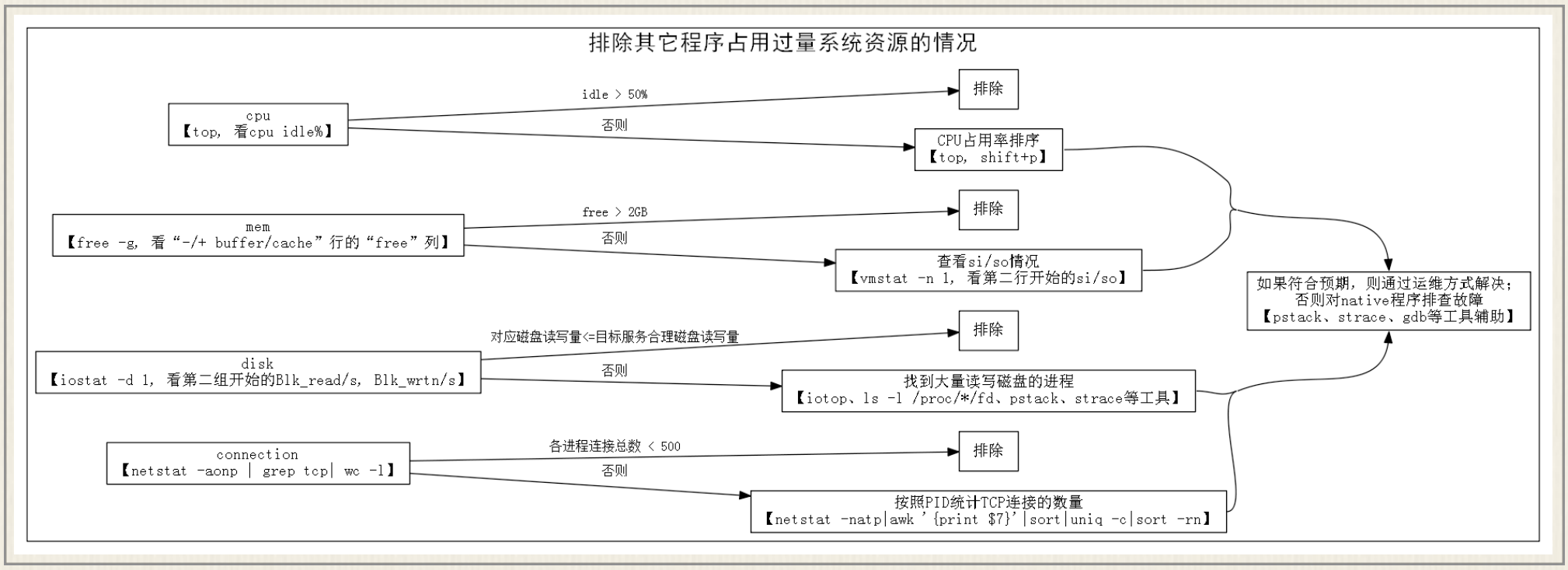
五、故障定位的步骤

我们采用“从外到内，逐步排除”的方式来定位故障：

1. 先排除其它程序过度占用系统资源的问题
2. 然后排除“目标服务”本身占用系统资源过度的问题
3. 最后观察目标服务内部的情况，排除掉各种常见故障类型。

对 于不能排除的方面，要根据该信息对应的“危险程度”来判断是应该“进一步深入”还是“暂时跳过”。例如“目标服务Java堆占用100%”这是一条危险程 度较高的信息，建议立即“进一步深入”。而对于“在CPU核数为8的机器上，其它程序偶然占用CPU达200%”这种危险程度不是很高的信息，则建议“暂 时跳过”。当然，有些具体情况还需要故障排查人员根据自己的经验做出判断。

第一步：排除其它程序占用过量系统资源的情况



图示：排除其它程序占用过量系统资源的情况

1. 运行【top】，检查CPU idle情况，如果发现idle较多（例如多余50%），则排除其它进程占用CPU过量的情况。

```
Tasks:  90 total,   1 running,  88 sleeping,   0 stopped,   1 zombie
Cpu(s):  0.6% us,  0.1% sy,  0.0% ni, 99.3% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:   3108716k total,  2235624k used,   873092k free,   112864k buffers
Swap:  2096472k total,  1048236k used,  1048236k free,   614464k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
```

如果idle较少，则按shift+p，将进程按照CPU占用率从高到低排序，逐一排查（见下面TIP）。

2. 运行【free -g】，检查剩余物理内存（“-/+ buffer/cache”行的“free”列）情况，如果发现剩余物理内存较多（例如剩余2GB以上），则排除占用物理内存过量的情况。

```
             total       used       free     shared    buffers     cached
Mem:             11         11          0          0          0          5
-/+ buffers/cache:          5          6
Swap:            16          0         15
```

如果剩余物理内存较少（例如剩余1GB以下），则运行【vmstat -n 1】检查si/so（换入换出）情况，

```
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0 283032  33276 499992 5827764    0    0    23    28    0    0  2  2 95  0  0
 6  0 283032  33284 499992 5827780    0    0     0    32  723 1112  0  0 100  0  0
 0  0 283032  33284 499992 5827788    0    0     0     0  688 1105  0  0 100  0  0
```

第 一行数值表示的是从系统启动到运行命令时的均值，我们忽略掉。从第二行开始，每一行的si/so表示该秒内si/so的block数。如果多行数值都为零，则可以排除物理内存不足的问题。如果数值较大（例如大于1000 blocks/sec，block的大小一般是1KB）则说明存在较明显的内存不足问题。我们可以运行【top】输入shift+m，将进程按照物理内存 占用（“RES”列）从大到小进行排序，然后对排前面的进程逐一排查（见下面TIP）。

3. 如果目标服务是磁盘I/O较重的程序，则用【iostat -d 1】，检查磁盘I/O情况。若“目标服务对应的磁盘”读写量在预估之内（预估要注意cache机制的影响），则排除其它进程占用磁盘I/O过量的问题。

```
Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda              17.64       239.05      1181.44 1813905030 8964665668
sdb               0.07         2.84         0.00   21554769       1376
sdc               0.92         7.81        17.91   59275773  135929264
sdd               0.39         0.69         4.37    5210701   33195568

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda               2.00         0.00       152.00          0        152
sdb               0.00         0.00         0.00          0          0
sdc               0.00         0.00         0.00          0          0
sdd               2.00         0.00        16.00          0         16
```

第一组数据是从该机器从开机以来的统计值。从第二组开始，都是每秒钟的统计值。通过【df】命令，可以看到Device与目录的关系。下图设备“sdb”就对应了目录“/disk2”。

```
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1             15235040  12094264   2354396  84% /
/dev/sda6            200985900 153876020  47105784  77% /disk1
/dev/sdb1            236545184 182632192  53908896  78% /disk2
/dev/sdc1            236545184  92644024 143897064  40% /disk3
/dev/sda5              1019208     51508    915092   6% /tmp
/dev/sda3              3050092    557544   2335112  20% /var
```

假如发现目标服务所在磁盘读写量明显超过推算值，则应该找到大量读写磁盘的进程（见下面TIP）

4. 运行【netstat -aonp | grep tcp| wc -l】查看各种状态的TCP连接数量和。如果总数较小（例如小于500），则排除连接数占用过多问题。

假如发现连接数较多，可以用【netstat -natp|awk '{print $7}'|sort|uniq -c|sort -rn】按照PID统计TCP连接的数量，然后对连接数较多的进程逐一排查（见下面TIP）。

TIP：如何"逐一排查"： 假如定位到是某个外部程序占用过量系统资源，则依据进程的功能和配置情况判断是否合乎预期。假如符合预期，则考虑将服务迁移到其他机器、修改程序运行的磁 盘、修改程序配置等方式解决。假如不符合预期，则可能是运行者对该程序不太了解或者是该程序发生了BUG。外部程序通常可能是Java程序也可能不是 Java程序，如果是Java程序，可以把它当作目标服务一样进行排查；而非Java程序具体排查方法超出了本文范围，列出三个工具供参考选用：

- 系统提供的调用栈的转储工具【pstack】，可以了解到程序中各个线程当前正在干什么，从而了解到什么逻辑占用了CPU、什么逻辑占用了磁盘等
- 系统提供的调用跟踪工具【strace】，可以侦测到程序中每个系统API调用的参数、返回值、调用时间等。从而确认程序与系统API交互是否正常等。

- 系统提供的调试器【gdb】，可以设置条件断点侦测某个系统函数调用的时候调用栈是什么样的。从而了解到什么逻辑不断在分配内存、什么逻辑不断在创建新连接等

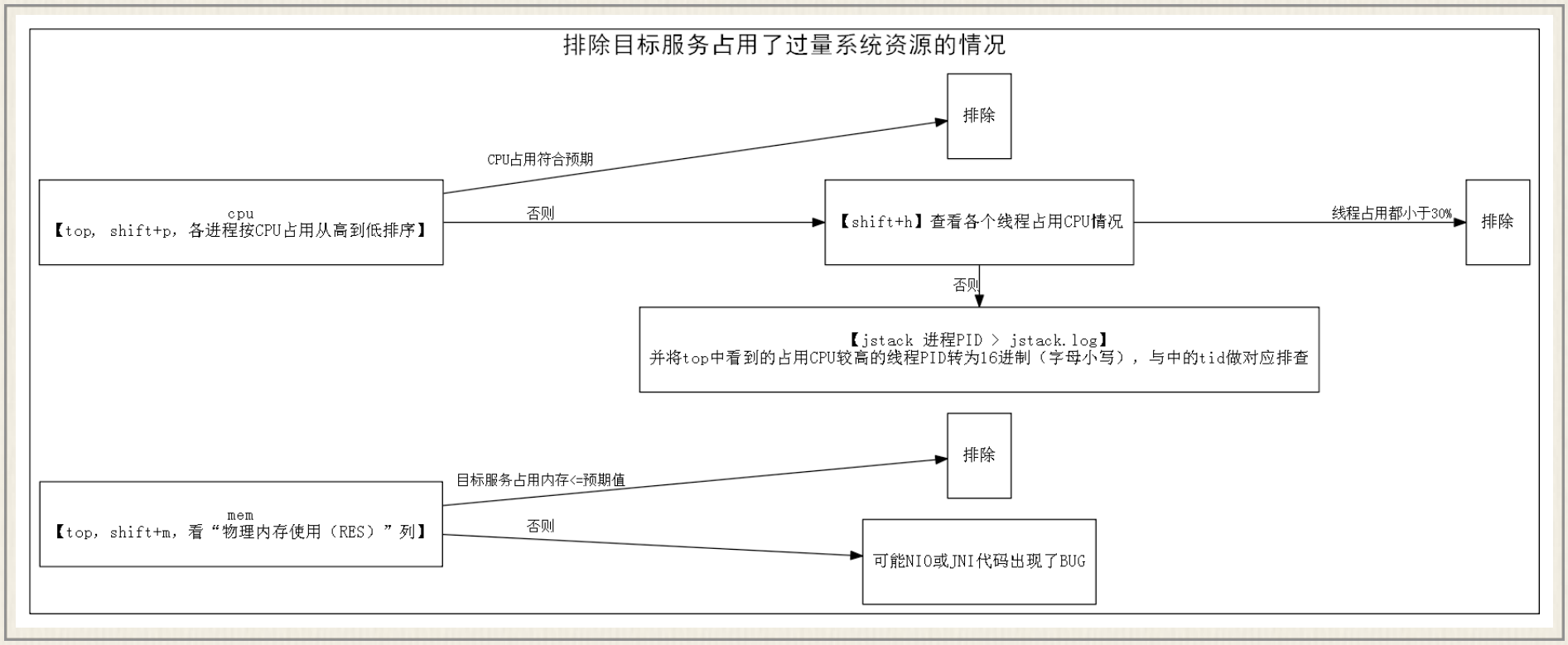
TIP：如何“找到大量读写磁盘的进程”：

1. 如果Linux系统比较新（kernel v2.6.20以上）可以使用iotop工具获知每个进程的io情况，较快地定位到读写磁盘较多的进程。

2. 通过【ls -l /proc/*/fd | grep 该设备映射装载到的文件系统路径】查看到哪个进程打开了该设备的文件，并根据进程身份、打开的文件名、文件大小等属性判断是否做了大量读写。

3..可以使用pstack取得进程的线程调用栈，或者strace跟踪磁盘读写API来帮助确认某个进程是否在做磁盘做大量读写

第二步：排除目标服务占用了过量系统资源的情况



图示：排除目标服务占用了过量系统资源的情况

1. 运行【top】，shift+p按照“CPU使用”从高到低的排序查看进程，假如目标服务占用的CPU较低（<100%，即小于一个核的计算量），或者符合经验预期，则排除目标服务CPU占用过高的问题。

假如目标服务占用的CPU较高（>100%，即大于一个核的计算量），则shift+h观察线程级别的CPU使用分布。

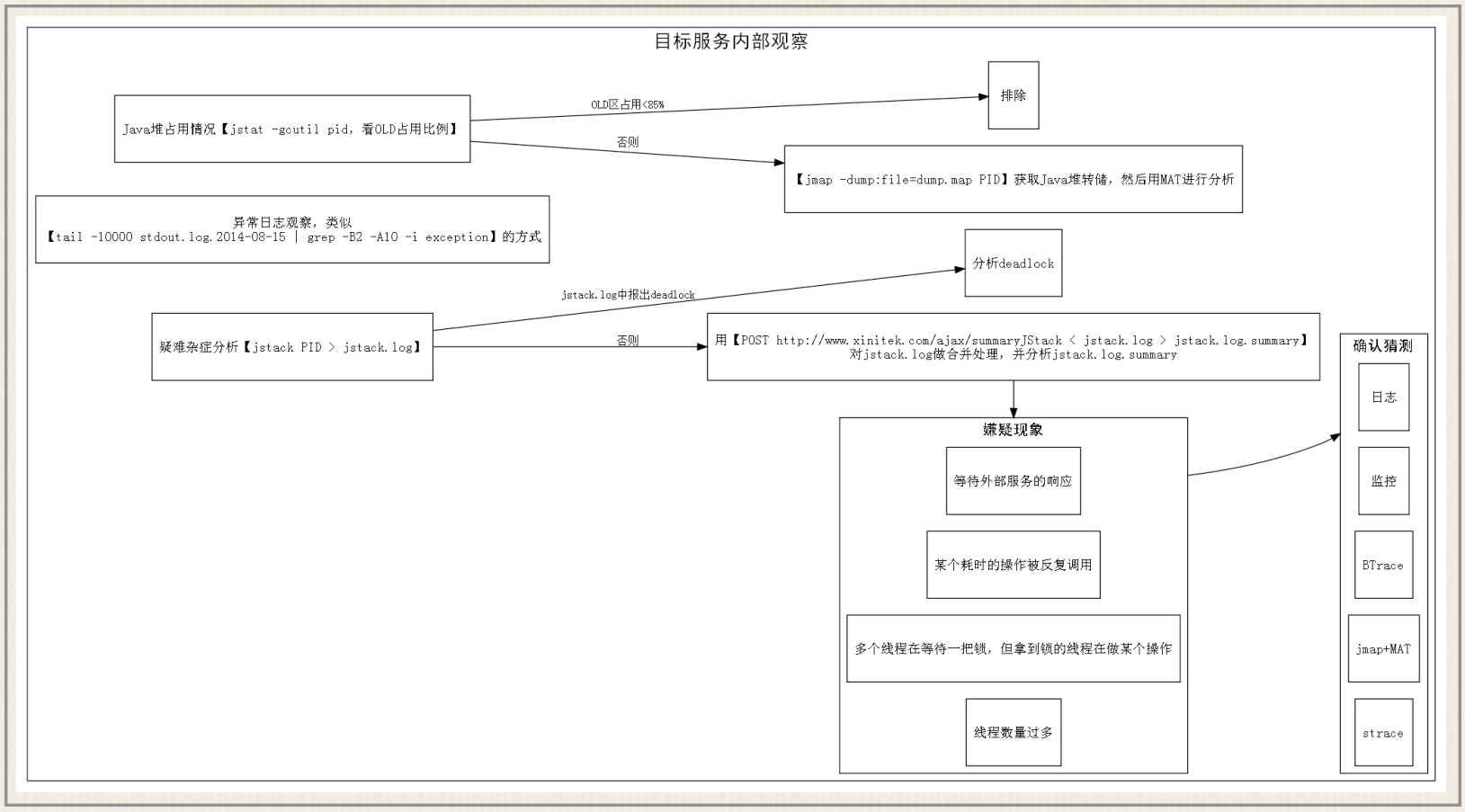
- 如果CPU使用分散到多个线程，而且每个线程占用都不算高（例如都<30%），则排除CPU占用过高的问题
- 如果CPU使用集中到一个或几个线程，而且很高（例如都>95%），则用【jstack pid > jstack.log】获取目标服务中线程调用栈的情况。top中看到的占用CPU较高的线程的PID转换成16进制（字母用小写），然后在 jstack.log中找到对应线程，检查其逻辑：
  - 假如对应线程是纯计算型任务（例如GC、正则匹配、数值计算等），则排除CPU占用过高的问题。当然如果这种线程占用CPU总量如果过多（例如占满了所有核），则需要对线程数量做控制（限制线程数 < CPU核数）。
  - 假如对应线程不是纯计算型任务（例如只是向其他服务请求一些数据，然后简单组合一下返回给用户等），而该线程CPU占用过高（>95%），则可能发生了异常。例如：死循环、数据结构过大等问题，确定具体原因的方法见下文“第三步：目标进程内部观察”。

2. 运行【top】，shift+m按照“物理内存使用(RES)”从高到低排序进程，评估目标服务占的内存量是否在预期之内。如果在预期之内，则排除目标服务Native内存占用过高的问题。

提示：由于Java进程中有Java级别的内存占用，也有Native级别的内存占用，所以Java进程的“物理内存使用(RES)”比“-Xmx参数指定的Java堆大小”大一些是正常的（例如1.5~2倍左右）。

假如“物理内存使用(RES)”超出预期较多（例如2倍以上），并且确定JNI逻辑不应该占用这么多内存，则可能是NIO或JNI代码出现了BUG。由于本文主要讨论的是Java级别的问题，所以对这种情况不做过多讨论。读者可以参考上文“TIP：如何逐一排查”进行native级别的调试。

第三步：目标服务内部观察



图示：目标服务内部观察

1. Java堆占用情况

用【jstat -gcutil pid】查看目标服务的OLD区占用比例，假如占用比例低于85%则排除Java堆占用比例过高的问题。

假如占用比例较高（例如超过98%），则服务存在Java堆占满的问题。这时候可以用jmap+mat进行分析定位内存中占用比例的情况（见下文TIP），从而较快地定位到Java堆满的原因。

TIP：用jmap+mat进行分析定位内存中占用比例的情况

先 通过【jmap -dump:file=dump.map pid】取得目标服务的Java堆转储，然后找一台空闲内存较大的机器在VNC中运行mat工具。mat工具中打开dump.map后，可以方便地分析内 存中什么对象引用了大量的对象（从逻辑意义上来说，就是该对象占用了多大比例的内存）。具体使用可以ca

2. 异常日志观察

通过类似【tail -10000 stdout.log.2014-08-15 | grep -B2 -A10 -i exception】这样的方式，可以查到日志中最近记录的异常。

3. 疑难杂症

用【jstack pid > jstack.log】获取目标服务中“锁情况”和“各线程调用栈”信息，并分析

- 检查jstack.log中是否有deadlock报出，如果没有则排除deadlock情况。

Found one Java-level deadlock:

==========

“Thread-0″:

waiting to lock monitor 0x1884337c (object 0x046ac698, a java.lang.Object),

which is held by “main”

“main”:

waiting to lock monitor 0x188426e4 (object 0x046ac6a0, a java.lang.Object),

which is held by “Thread-0″

Java stack information for the threads listed above:

==========

“Thread-0″:

at LockProblem$T2.run(LockProblem.java:14)

- waiting to lock <0x046ac698> (a java.lang.Object)

- locked <0x046ac6a0> (a java.lang.Object)

“main”:

at LockProblem.main(LockProblem.java:25)

- waiting to lock <0x046ac6a0> (a java.lang.Object)

- locked <0x046ac698> (a java.lang.Object)

Found 1 deadlock.

如果发现deadlock则则根据jstack.log中的提示定位到对应代码逻辑。

- 用【POST http://www.xinitek.com/ajax/summaryJStack < jstack.log > jstack.log.summary】对jstack.log做合并处理，然后继续分析故障所在。

通过jstack.log.summary中的情况，我们可以较迅速地定位到一些嫌疑点，并可以猜测其故障引起的原因（后文有jstack.log.summary情况举例供参考）

情况	嫌疑点	猜测原因
线程数量过多	某种线程数量过多	运行环境中“限制线程数量”的机制失效
多个线程在等待一把锁，但拿到锁的线程在做某个操作	拿到这把锁的线程在做网络connect操作	被connect的服务异常
	拿到锁的线程在做数据结构遍历操作	该数据结构过大或被破坏
某个耗时的操作被反复调用	某个应当被缓存的对象多次被创建	对象池的配置错误
等待外部服务的响应	很多线程都在等待外部服务的响应	该外部服务故障
	很多线程都在等待FutureTask完成，而FutureTask在等待外部服务的响应	该外部服务故障

猜测了原因后，可以通过日志检查、监控检查、用测试程序尝试复现等方式确认猜测是否正确。如果需要更细致的证据来确认，可以通过BTrace、strace、jmap+MAT等工具进行分析，最终确认问题所在。

下面简单介绍下这几个工具：

BTrace：用于监测Java级别的方法调用情况。可以对运行中的Java虚拟机插入调试代码，从而确认方法每次调用的参数、返回值、花费时间等。第三方免费工具。

strace：用于监视系统调用情况。可以得到每次系统调用的参数、返回值、耗费时间等。Linux自带工具。

jmap+MAT： 用于查看Java级别内存情况。jmap是JDK自带工具，可以将Java程序的Java堆转储到数据文件中；MAT是eclipse.org上提供的一 个工具，可以检查jmap转储数据文件中的数据。结合这两个工具，我们可以非常容易地看到Java程序内存中所有对象及其属性。

TIP：jstack.log.summary情况举例

1. 某种线程数量过多

1000 threads at

“Timer-0″ prio=6 tid=0x189e3800 nid=0x34e0 in Object.wait() [0x18c2f000]

java.lang.Thread.State: TIMED_WAITING (on object monitor)

at java.lang.Object.wait(Native Method)

at java.util.TimerThread.mainLoop(Timer.java:552)

- locked [***] (a java.util.TaskQueue)

at java.util.TimerThread.run(Timer.java:505)

2. 多个线程在等待一把锁，但拿到锁的线程在做数据结构遍历操作

38 threads at

“Thread-44″ prio=6 tid=0×18981800 nid=0x3a08 waiting for monitor entry [0x1a85f000]

java.lang.Thread.State: BLOCKED (on object monitor)

at SlowAction$Users.run(SlowAction.java:15)

- waiting to lock [***] (a java.lang.Object)

1 threads at

“Thread-3″ prio=6 tid=0x1894f400 nid=0×3954 runnable [0x18d1f000]

java.lang.Thread.State: RUNNABLE

at java.util.LinkedList.indexOf(LinkedList.java:603)

at java.util.LinkedList.contains(LinkedList.java:315)

at SlowAction$Users.run(SlowAction.java:18)

- locked [***] (a java.lang.Object)

3. 某个应当被缓存的对象多次被创建（数据库连接）

99 threads at

“resin-tcp-connection-*:3231-321″ daemon prio=10 tid=0x000000004dc43800 nid=0x65f5 waiting for monitor entry [0x00000000507ff000]

java.lang.Thread.State: BLOCKED (on object monitor)

at org.apache.commons.dbcp.PoolableConnectionFactory.makeObject(PoolableConnectionFactory.java:290)

- waiting to lock <0x00000000b26ee8a8> (a org.apache.commons.dbcp.PoolableConnectionFactory)

at org.apache.commons.pool.impl.GenericObjectPool.borrowObject(GenericObjectPool.java:771)

at org.apache.commons.dbcp.PoolingDataSource.getConnection(PoolingDataSource.java:95)

…

1 threads at

“resin-tcp-connection-*:3231-149″ daemon prio=10 tid=0x000000004d67e800 nid=0x66d7 runnable [0x0000000005180f000]

java.lang.Thread.State: RUNNABLE

…

at org.apache.commons.dbcp.DriverManagerConnectionFactory.createConnection(DriverManagerConnectionFactory.java:46)

at org.apache.commons.dbcp.PoolableConnectionFactory.makeObject(PoolableConnectionFactory.java:290)

- locked <0x00000000b26ee8a8> (a org.apache.commons.dbcp.PoolableConnectionFactory)

at org.apache.commons.pool.impl.GenericObjectPool.borrowObject(GenericObjectPool.java:771)

at org.apache.commons.dbcp.PoolingDataSource.getConnection(PoolingDataSource.java:95)

at …

4. 很多线程都在等待外部服务的响应

100 threads at

“Thread-0″ prio=6 tid=0x189cdc00 nid=0×2904 runnable [0x18d5f000]
java.lang.Thread.State: RUNNABLE
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.read(SocketInputStream.java:150)
at java.net.SocketInputStream.read(SocketInputStream.java:121)
…
at RequestingService$RPCThread.run(RequestingService.java:24)

5. 很多线程都在等待FutureTask完成，而FutureTask在等待外部服务的响应

100 threads at

“Thread-0″ prio=6 tid=0×18861000 nid=0x38b0 waiting on condition [0x1951f000]

java.lang.Thread.State: WAITING (parking)

at sun.misc.Unsafe.park(Native Method)

- parking to wait for [***] (a java.util.concurrent.FutureTask$Sync)

at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)

at java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:834)

at java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireSharedInterruptibly(AbstractQueuedSynchronizer.java:994)

at
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInter
ruptibly(AbstractQueuedSynchronizer.java:1303)

at java.util.concurrent.FutureTask$Sync.innerGet(FutureTask.java:248)

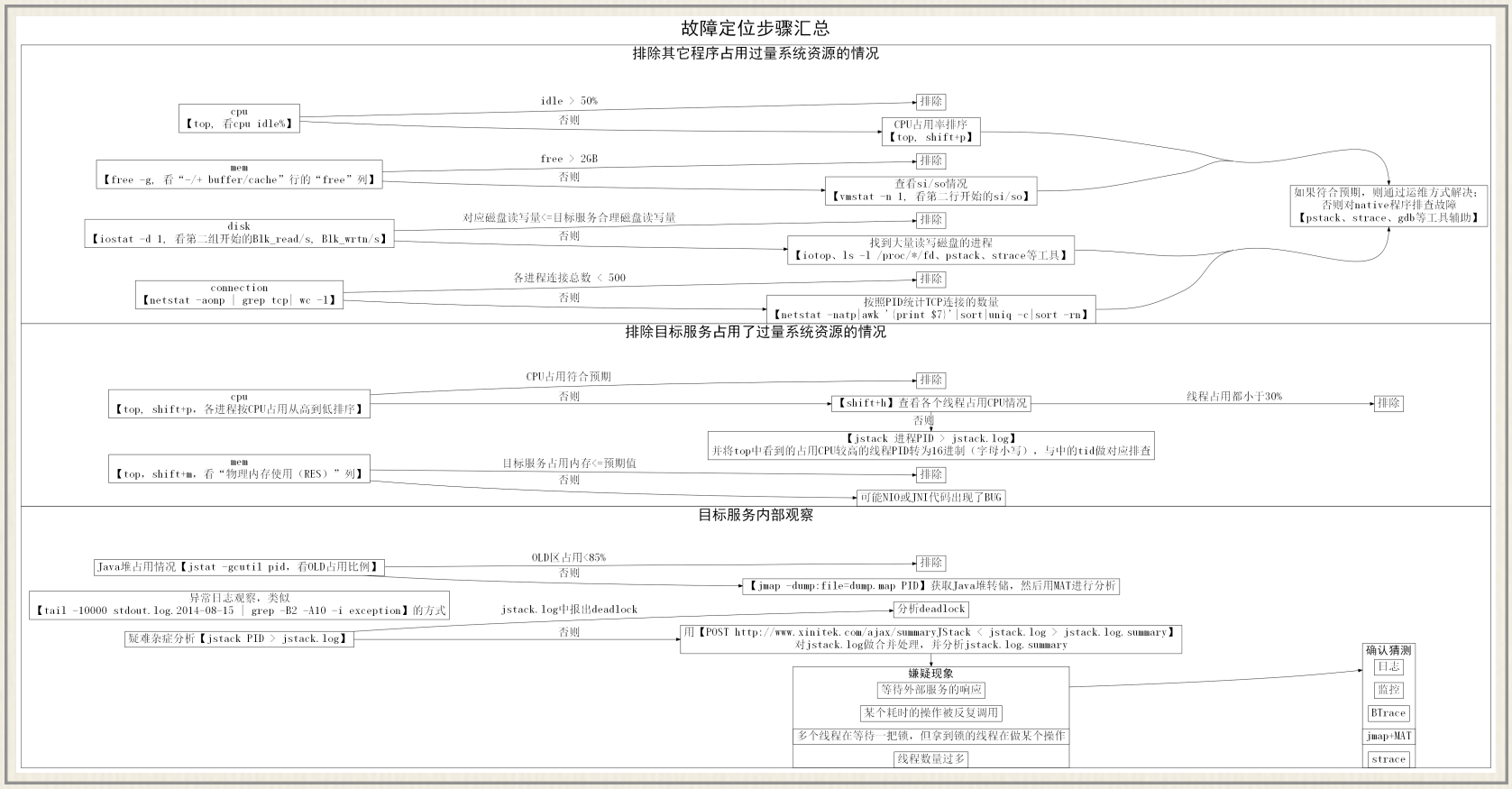
at java.util.concurrent.FutureTask.get(FutureTask.java:111)

at IndirectWait$MyThread.run(IndirectWait.java:51)

100 threads at

“pool-1-thread-1″ prio=6 tid=0x188fc000 nid=0×2834 runnable
[0x1d71f000]
java.lang.Thread.State: RUNNABLE
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.read(SocketInputStream.java:150)
at java.net.SocketInputStream.read(SocketInputStream.java:121)
…
at IndirectWait.request(IndirectWait.java:23)
at IndirectWait$MyThread$1.call(IndirectWait.java:46)
at IndirectWait$MyThread$1.call(IndirectWait.java:1)
at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
at java.util.concurrent.FutureTask.run(FutureTask.java:166)
at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.j
ava:1110)
at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.
java:603)
at java.lang.Thread.run(Thread.java:722)

为方便读者使用，将故障定位三个步骤的图合并如下:



图示：故障定位步骤汇总

故障定位是一个较复杂和需要经验的过程，如果现在故障正在发生，对于分析经验不很多的开发或运维人员，有什么简单的操作步骤记录下需要的信息吗？下面提供一个

六、给运维人员的简单步骤

如果事发突然且不能留着现场太久，要求运维人员：

1. top: 记录cpu idle%。如果发现cpu占用过高，则c, shift+h, shift + p查看线程占用CPU情况，并记录

2. free: 查看内存情况，如果剩余量较小，则top中shift+m查看内存占用情况，并记录

3. 如果top中发现占用资源较多的进程名称（例如java这样的通用名称）不太能说明进程身份，则要用ps xuf | grep java等方式记录下具体进程的身份

4. 取jstack结果。假如取不到，尝试加/F

jstack命令：jstack PID > jstack.log

5. jstat查看OLD区占用率。如果占用率到达或接近100%，则jmap取结果。假如取不到，尝试加/F

jstat命令： jstat -gcutil PID

S0 S1 E O P YGC YGCT FGC FGCT GCT

0.00 21.35 88.01 97.35 59.89 111461 1904.894 1458 291.369 2196.263

jmap命令： jmap -dump:file=dump.map PID

6. 重启服务

原文链接：http://techblog.youdao.com/?p=961

MySQL原生HA方案 – Fabric体验之旅

作者：盘古

还在为MySQL HA方案的选型头疼吗？现在不用了，自从2014年5月28日Oracle发布Fabric之后，一切都变得如此简单。因为是原生的官方产品，可以放心使 用，由于这款产品大力的增强了HA效率，可以看出Oracle对云计算的支持力度，说明Oracle这个大象也可跳舞，而且还挺灵活的。

官方对Fabric的介绍主要是它提供了MySQL的HA和Sharding方案，本文主要讨论Fabric在MySQL HA方面的表现以及搭建部署流程。我的团队试着搭建了一下，简直无法再简单了，这对于DBA来说绝对是一个大福音，这个产品在接下来的几年中必然会被大量 应用到生产环境中去，我的团队对这个产品的未来充满信心。

接下来我们将循序渐进的带领大家感受Fabric带来的乐趣。

一、实验环境

本例的实验环境是在一台CentOS主机中做的，机器上有3个MySQL实例，分别是3306、3691和3692，我们要做的就是用这3个实例达成HA效果

功能	IP	PORT
backing store	10.165.17.175	3306
MySQL 主		3691
MySQL 备		3692
Fabric 管理进程		32274

二、下载

Fabric目前是被打包到了MySQL Utilities中，所以大家下载MySQL Utilities就可以了，下载地址是： http://dev.mysql.com/downloads/utilities/，目前版本是：1.4.4，在本例中我们下载的是mysql-utilities-1.4.4-1.el6.noarch.rpm。

三、安装MySQL Utilities

rpm包的安装还是挺简单的，具体如下：

rpm -ivh mysql-utilities-1.4.4-1.el6.noarch.rpm

Preparing...
[100%]

1:mysql-utilities
[100%]

[root@

装完后执行

mysqlfabric

如果有回显说明安装完毕。

四、建立Backing Store帐号

Backing Store用于存储整个HA集群的服务器等相关配置，它需要一个MySQL实例来存储这些信息，这个实例的版本需要跟其它在HA中的

MySQL实例版本保持一致，而且必须是5.6.10及更高的版本，我们在本例中选择3306实例来使用。

首先，你需要一个帐号来连接Backing Store的MySQL实例，这个帐号需要有对fabric数据库的管理员级权限，我们在3306端口的实例上建帐号，具体如下：

```
CREATE USER 'fabric'@'10.165.17.175' IDENTIFIED BY 'secret';
GRANT ALL ON fabric.* TO 'fabric'@'10.165.17.175';
```

五、Fabric配置文件

Fabric配置文件默认位置是：/etc/mysql/fabric.cfg

修改其中的[storage]部分，具体如下：

```
[storage]
auth_plugin = mysql_native_password
database = fabric
user = fabric
address = 10.165.17.175:3306
connection_delay = 1
connection_timeout = 6
password = secret
connection_attempts = 6
```

其中address = 10.165.17.175:3306是Backing Store的MySQL实例，password = secret是上一步中建立连接fabric数据库的用户密码。

修改其中的[servers]部分，具体如下：

[servers]

password = secret

user = fabric

其中是password = secret 是HA环境中各实例的连接密码。

六、填充Backing Store信息

我们通过Fabric来填充3306端口实例中的fabric数据库，具体如下：

mysqlfabric manage setup

[INFO] 1408115689.486792 - MainThread - Initializing persister: user (fabric), server (10.165.17.175:3306), database (fabric).

Finishing initial setup

==========

Password for admin user is not yet set.

Password for admin/xmlrpc:

Repeat Password:

Password set.

操作期间会提示Fabric的管理员帐户admin没有设置密码，咱们按提示将密码设置成admin就可以了。

我们再查看3306端口的实例里面发生了什么变化，具体如下：

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| 51linux.net        |
| fabric             |
| mysql              |
| performance_schema |
+--------------------+
5 rows in set (0.00 sec)

mysql>
```

可以看到多了一个fabric数据库，它里面存储的就是Fabric的一些配置信息。

七、配置HA中主从MySQL节点帐号

本例中3691和3692实例是需要做成HA的，它们也要建个管理员权限的帐号，注意，帐号名也要跟3306实例保持一致，也需要是fabric，具体如下：

```
CREATE USER 'fabric'@'10.165.17.175' IDENTIFIED BY 'secret';
GRANT ALL ON *.* TO 'fabric'@'10.165.17.175';
```

同时，由于fabric是基于GTID主从复制，所以这些实例中必须要启用GTID，它们的配置文件要有这些参数：

log-bin

gtid-mode=ON

enforce-gtid-consistency

log_slave_updates

八、启动fabric

我们用下面的命令来启动fabric：

mysqlfabric manage start

[INFO] 1408116209.229260 - MainThread - Initializing persister: user (fabric), server (10.165.17.175:3306), database (fabric).

[INFO] 1408116209.233982 - MainThread - Loading Services.

[INFO] 1408116209.253620 - MainThread - Fabric node starting.

[INFO] 1408116209.261853 - MainThread - Starting Executor.

[INFO] 1408116209.262001 - MainThread - Setting 5 executor(s).

[INFO] 1408116209.262691 - Executor-0 - Started.

[INFO] 1408116209.264825 - Executor-1 - Started.

[INFO] 1408116209.266648 - Executor-2 - Started.

[INFO] 1408116209.268395 - Executor-3 - Started.

[INFO] 1408116209.269961 - MainThread - Executor started.

[INFO] 1408116209.273374 - MainThread - Starting failure detector.

[INFO] 1408116209.274144 - Executor-4 - Started.

[INFO] 1408116209.275816 - XML-RPC-Server - XML-RPC protocol server ('127.0.0.1', 32274) started.

[INFO] 1408116209.276112 - XML-RPC-Server - Setting 5 XML-RPC session(s).

[INFO] 1408116209.276654 - XML-RPC-Session-0 - Started XML-RPC-Session.

[INFO] 1408116209.278426 - XML-RPC-Session-1 - Started XML-RPC-Session.

[INFO] 1408116209.280368 - XML-RPC-Session-2 - Started XML-RPC-Session.

[INFO] 1408116209.281599 - XML-RPC-Session-3 - Started XML-RPC-Session.

[INFO] 1408116209.282699 - XML-RPC-Session-4 - Started XML-RPC-Session.

九、建立HA服务器组

这个HA服务器组，用于把参与HA的所有MySQL实例都填加进来：

mysqlfabric group create my_group

Password for admin:

Procedure :

{ uuid = 292621fd-cddc-4cbb-8c0d-d8a264156679,

finished = True,

success = True,

return = True,

activities =

}

这样我们就创建了一个组名为my_group的HA服务器组。

十、填加HA组的成员

我们首先填加3691，具体如下：

mysqlfabric group add my_group 10.165.17.175:3691

Password for admin:

Procedure :

{ uuid = 8d1c11f8-adc4-4321-8307-6296caeb07c1,

finished = True,

success = True,

return = True,

activities =

}

接下来填3692，具体如下：

mysqlfabric group add my_group 10.165.17.175:3692

Password for admin:

Procedure :

{ uuid = b1fa3cb9-b86f-4b1a-88cb-e84babb2ab02,

finished = True,

success = True,

return = True,

activities =

}

如果屏幕回显示无error，那么说明成功填加了成员。我们也可以查看一下my_group里面的成员信息，具体如下：

mysqlfabric group lookup_servers my_group

Password for admin:

Command :

{ success = True

return = [{'status': 'SECONDARY', 'server_uuid': '6914a176-2370-11e4-af48-00163e004141', 'mode': 'READ_ONLY', 'weight': 1.0, 'address': '10.165.17.175:3691'}, {'status': 'SECONDARY', 'server_uuid': 'a8a69428-2366-11e4-af09-00163e004141', 'mode': 'READ_ONLY', 'weight': 1.0, 'address': '10.165.17.175:3692'}]

activities =

}

大家可以看到，这2个实例都不是PRIMARY，说明刚刚搭建完的环境，系统是不会选举出PRIMARY的。

十一、选举一个主库

选举的方法也非常简单，具体如下：

mysqlfabric group promote my_group

Password for admin:

Procedure :

{ uuid = 529380b9-10ef-409f-a1a9-9430ab9845a3,

finished = True,

success = True,

return = True,

activities =

}

可见执行成功了，并没有返回error。

接下来我们再次验证一下HA集群中各服务器情况。

mysqlfabric group lookup_servers my_group

Password for admin:

Command :

{ success = True

return = [{'status': 'SECONDARY', 'server_uuid': '6914a176-2370-11e4-af48-00163e004141', 'mode': 'READ_ONLY', 'weight': 1.0, 'address': '10.165.17.175:3691, {'status': 'PRIMARY', 'server_uuid': 'a8a69428-2366-11e4-af09-00163e004141', 'mode': 'READ_WRITE', 'weight': 1.0, 'address': '10.165.17.175:3692}]

```
  activities  =
}
```

可见Fabric已经随机选举了一个Master角色。

十二、激活故障自动切换

即使Fabric选出了Master角色，但当这个Master宕机时，Fabric并不会自动将Secondary角色切换成Master角色，所以我们需要将HA配置成可以自动切换角色的样子，具体如下：

```
mysqlfabric group activate my_group
Password for admin:
Procedure :
{ uuid        = 518b7dad-06a4-45a8-bfd5-241396706b88,
  finished    = True,
  success     = True,
  return      = True,
  activities  =
}
```

当然，我们也可以依据具体需求取消Fabric故障自动切换。

十三、测试HA

在这个实验中，我们将3691实例停止，再看看Fabric的状态：

mysqlfabric group lookup_servers my_group

Password for admin:

Command :

{ success = True

return = [{'status': 'PRIMARY', 'server_uuid': '6914a176-2370-11e4-af48-00163e004141', 'mode': 'READ_WRITE', 'weight': 1.0, 'address': '10.165.17.175:3691}, {'status': 'FAULTY', 'server_uuid': 'a8a69428-2366-11e4-af09-00163e004141', 'mode': 'READ_WRITE', 'weight': 1.0, 'address': '10.165.17.175:3692}]

activities =

}

其中3692实例的状态已经变成了“FAULTY”，可以看出Fabric自动检测到了这个故障，并且选举了slave重新当了primary角色。我不得不说就是这个功能，是它吸引我的原因之一。

十四、后续学习

关于后续的学习，大家要看看官网的用户手册，里面还有很多HA维护的方法，如增减节点等问题，同时目前Fabric也提供了phtyon和Java 的API，可以供软件开发人员直接使用，以后的软件开发人员，不是再直接连接到MySQL实例，而是连接到Fabric，由Fabric来统一分发请求， 这有些象MySQL Proxy，但它的应用前景要比MySQL Proxy更宽更广。

原文链接：http://www.csdn.net/article/2014-08-20/2821300

网易OpenStack部署运维实战

作者：张晓龙、王盼、管强、高田田

OpenStack 简介

OpenStack 是一个开源的 IaaS 实现，它由一些相互关联的子项目组成，主要包括计算、存储、网络。由于以 Apache 协议发布，自 2010 年项目成立以来，超过 200 个公司加入了 OpenStack 项目，其中包括 AT&T、AMD、Cisco、Dell、IBM、Intel、Red Hat 等。目前参与 OpenStack 项目的开发人员有 17,000+，来自 139 个国家，这一数字还在不断增长中。

OpenStack 兼容一部分 AWS 接口，同时为了提供更强大的功能，也提供 OpenStack 风格的接口（RESTFul API）。和其他开源 IaaS 相比，架构上松耦合、高可扩展、分布式、纯 Python 实现，以及友好活跃的社区使其大受欢迎，每半年一次的开发峰会也吸引了来自全世界的开发者、供应商和客户。

OpenStack 的主要子项目有：

- Compute（Nova）提供计算虚拟化服务，是 OpenStack 的核心，负责管理和创建虚拟机。它被设计成方便扩展，支持多种虚拟化技术，并且可以部署在标准硬件上。
- Object Storage（Swift） 提供对象存储服务，是一个分布式，可扩展，多副本的存储系统。
- Block Storage（Cinder），提供块存储服务，为 OpenStack 的虚拟机提供持久的块级存储设备。支持多种存储后端，包括 Ceph，EMC 等。
- Networking（Neutron ）提供网络虚拟化服务，是一个可拔插，可扩展，API 驱动的服务。

- Dashboard 提供了一个图形控制台服务，让用户方便地访问，使用和维护 OpenStack 中的资源。
- Image（glance）提供镜像服务，它旨在发现，注册和交付虚拟机磁盘和镜像。支持多种后端。
- Telemetry（Ceilometer）提供用量统计服务，通过它可以方便地实现 OpenStack 计费功能。
- Orchestration（Heat）整合了 OpenStack 中的众多组件，类似 AWS 的 CloudFormation，让用户能够通过模板来管理资源。
- Database（Trove）基于 OpenStack 构建的 database-as-a-service。

网易私有云使用了 Nova、Glance、Keystone、Neutron 这 4 个组件。

网易私有云平台概况

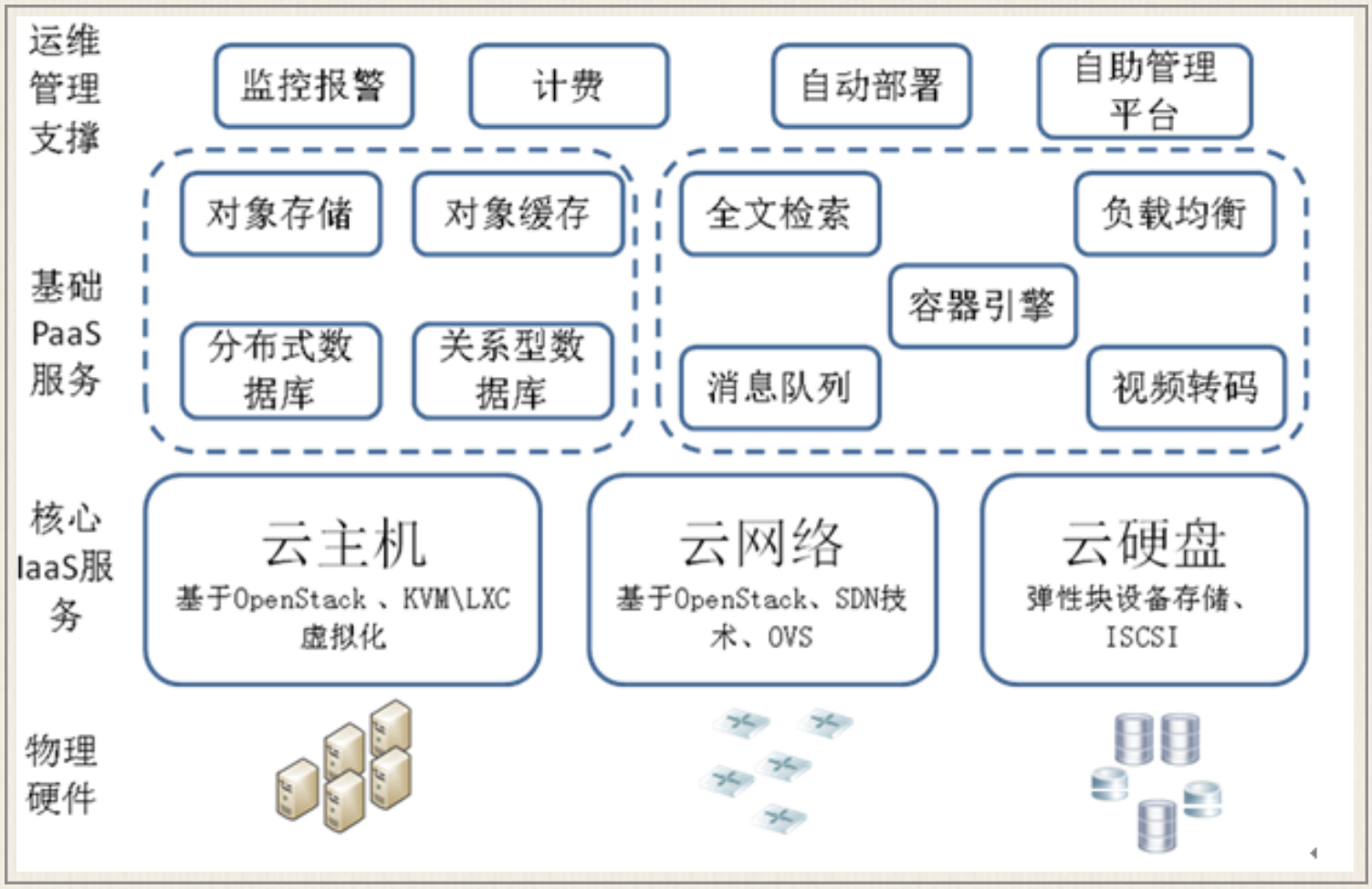


图 1.网易私有云架构

网易私有云平台由网易杭州研究院负责研发，主要提供基础设施资源、数据存储处理、应用开发部署、运维管理等功能以满足公司产品测试/上线的需求。

图 1 展示了网易私有云平台的整体架构。整个私有云平台可分为三大类服务：核心基础设施服务（IaaS）、基础平台服务（PaaS）以及运维管理支撑服务，目前 一共包括了：云主机（虚拟机）、云网络、云硬盘、对象存储、对象缓存、关系型数据库、分布式数据库、全文检索、消息队列、视频转码、负载均衡、容器引擎、 云计费、云监控、管理平台等 15 个服务。网易私有云平台充分利用云计算开源的最新成果，我们基于 OpenStack 社区的 keystone、glance、nova、neutron 组件研发部署了云主机和云网络服务。

为了与网易私有云平台其他服务（云硬盘、 云监控、云计费等）深度整合以及满足公司产品使用和运维管理的特定需求，我们团队在社区 OpenStack 版本的基础上独立研发了包括：云主机资源质量保障（计算、存储、网络 QoS）、镜像分块存储、云主机心跳上报、flat-dhcp 模式下租户内网隔离等 20 多个新功能。同时，我们团队在日常运维 OpenStack 以及升级社区新版本中，也总结了一些部署、运维规范以及升级经验。两年多来，网易私有云平台 OpenStack 团队的研发秉承开源、开放的理念，始终遵循"来源社区，回馈社区"的原则。在免费享受 OpenStack 社区不断研发新功能以及修复 bug 的同时，我们团队也积极向社区做自己的贡献，从而帮助 OpenStack 社区的发展壮大。两年来，我们团队一共向社区提交新功能开发/bug 修复的 commits 近 100 个，修复社区 bug 50 多个，这些社区贡献涉及 OpenStack 的 Essex、Folsom、Havana、Icehouse、Juno 等版本。

得益于 OpenStack 的日益稳定成熟，私有云平台目前已经稳定运行了 2 年多时间，为网易公司多达 30 个互联网和游戏产品提供服务。从应用的效果来看，基于 OpenStack 研发的网易私有云平台已经达到了以下目标：

1. 提高了公司基础设施资源利用率，从而降低了硬件成本。以物理服务器 CPU 利用率为例，私有云平台将 CPU 平均利用率从不到 10% 提升到 50%。

2. 提高了基础设施资源管理与运维自动化水平，从而降低了运维成本。借助于 Web 自助式的资源申请和分配方式以及云平台自动部署服务，系统运维人员减少了 50%。

3. 提高了基础设施资源使用弹性，从而增强了产品业务波动的适应能力。利用虚拟化技术将物理基础设施做成虚拟资源池，通过有效的容量规划以及按需使用，私有云平台可以很好适应产品突发业务。

网易 OpenStack 部署参考方案介绍

在具体的生产环境中，我们为了兼顾性能和可靠性，keystone 后端使用 Mysql 存储用户信息，使用 memcache 存放 token。为了减少对 keystone 的访问压力，所有服务（nova，glance，neutron）的 keystoneclient 均配置使用 memcache 作为 token 的缓存。

由于网易私有云需要部署在多个机房之中，每个机房之间在地理位置上自然隔离，这对上层的应用来说是天然的容灾方法。另外，为了满足私有云的功能和运维需 求，网易私有云需要同时支持两种网络模式：nova-network 和 neu-
tron。针对这些需求，我们提出了一个面向企业级的多区域部署方案，如图 2 所示。从整体上看，多个区域之间的部署相对独立，但可通过内网实现互通，每个区域中包括了一个完整的 OpenStack 部署，所以可以使用独立的镜像服务和独立的网络模式，例如区域 A 使用 nova-network，区域 B 使用 neutron，互不影响，另外为了实现用户的单点登录，区域之间共享了 key-stone，区域的划分依据主要是网络模式和地理位置。

和典型 OpenStack 部署将硬件划分为计算节点和控制节点不同的是，为了充分利用硬件资源，我们努力把部署设计成对称的，即任意一个节点下线对整体服务不会照成影响。因此我们 将硬件分为两类：计算节点，控制计算节点。计算节点部署 nova-network，nova-compute，nova-api-metadata，nova-api-os-compute。控制计算节点除了 计算节点的服务外还部署了 nova-scheduler，nova-novncproxy，nova-consoleauth，glance-api，glance- registry 和 keystone，如图 3 所示。

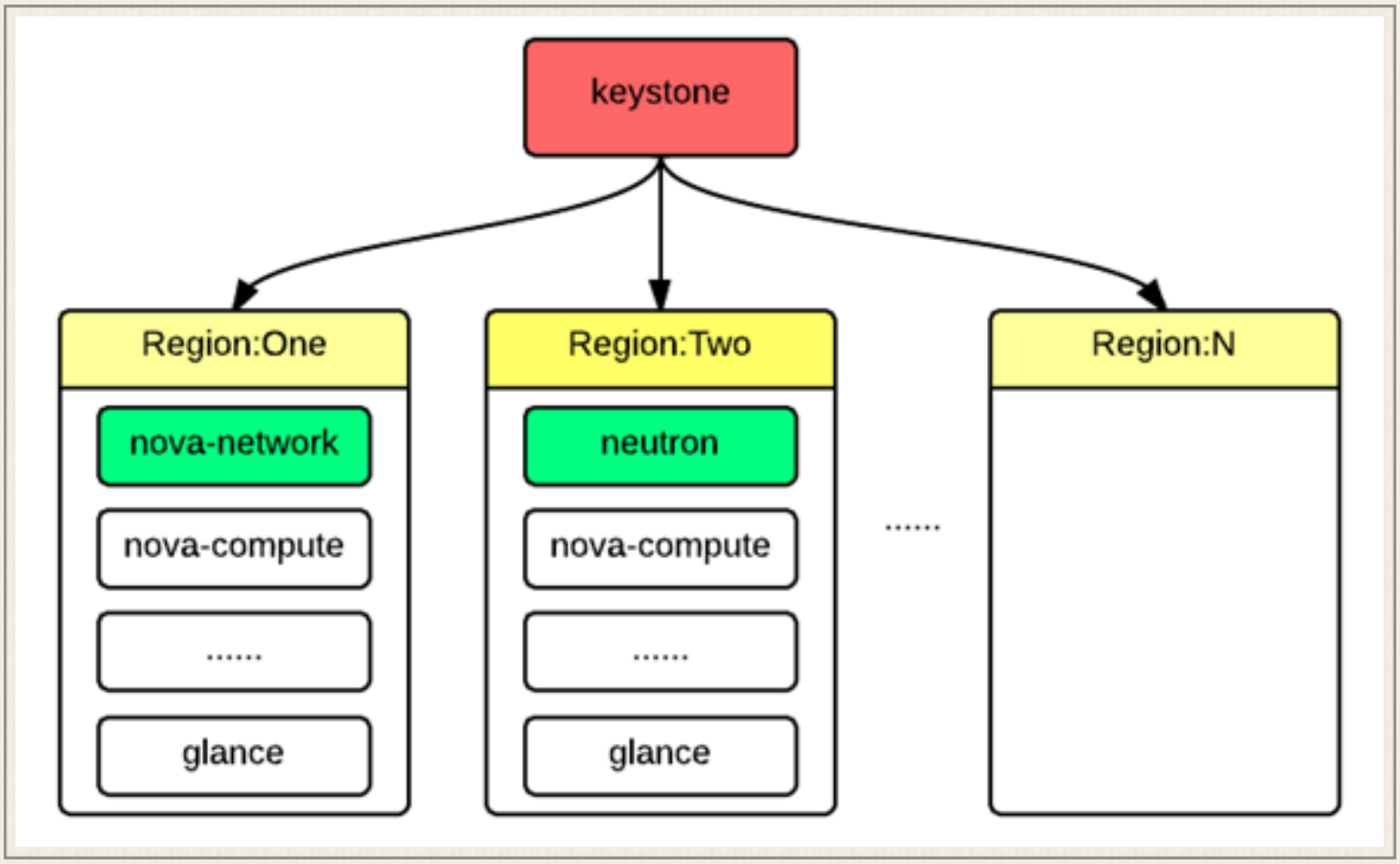
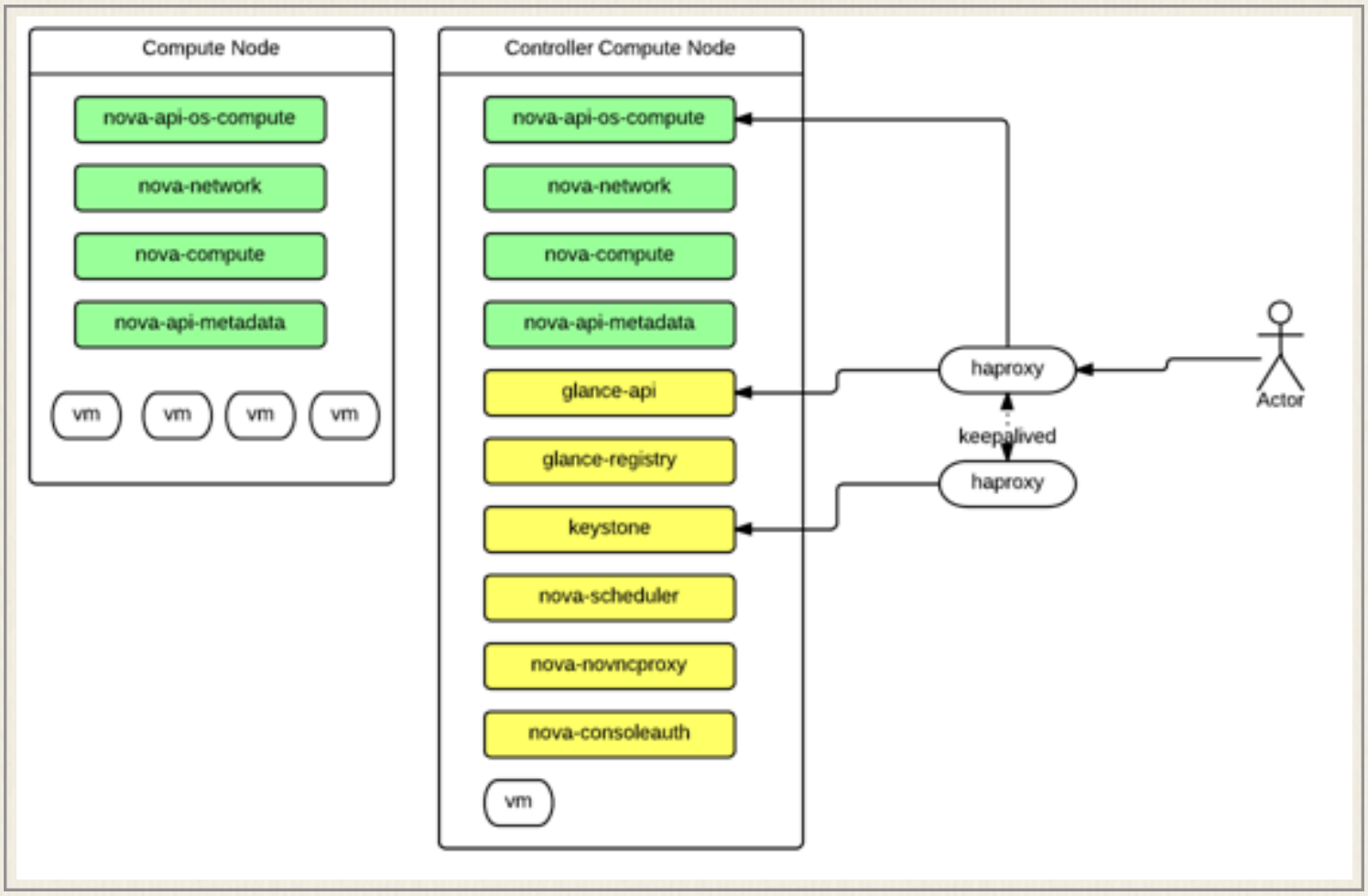


图 2.多区域部署方法

对外提供 API 的服务有 nova-api-os-compute，nova-novncproxy ,glance-api ，keystone 。这类服务的特点是无状态，可以方便地横向扩展，故此类服务均部署在负载均衡 HAProxy 之后，并且使用 Keepalived 做高可用。为了保证服务质量和便于维护，我们没有使用 nova-api，而是分为 nova-api-os-compute 和 nova-api-metadata 分别管理。外部依赖方面，网易私有云部署了高可用 RabbitMQ 集群和主备 MySQL，以及 memcache 集群。

网络规划方面，网易私有云主要使用 nova-network 的 FlatDHCPManager+multi-host 网络模式，并划分了多个 Vlan，分别用于虚拟机 fixed-ip 网络、内网浮动 IP 网络、外网网络。

运维上使用网易自主研发的运维平台做监控和报警，功能类似 Nagios，但是更加强大。其中较重要的监控报警包括日志监控和进程监控。日志监控保证服务发生异常时第一时间发现，进程监控保证服务正常运行。另外网易私有云使用 Puppet 做自动部署，以及使用 StackTach 帮助定位 bug。



OpenStack 各组件配置

OpenStack Havana 的配置项成百上千，大部分配置项都是可以使用默认值的，否则光是理解这么多的配置项的含义就足以让运维人员崩溃，尤其是对那些并不熟悉源码的运维人员来说 更是如此。下文将列举若干网易私有云中较关键的配置项，并解释它们如何影响到服务的功能，安全性，以及性能等问题。

Nova 关键配置

my_ip = 内网地址

此项是用来生成宿主机上的 nova metadata api 请求转发 iptables 规则，如果配置不当，会导致虚拟机内部无法通过 169.254.169.254 这个 IP 获取 ec2/OpenStack metadata 信息；生成的 iptable 规则形如：

*-A nova-network-PREROUTING -d 169.254.169.254/32 -p tcp -m tcp --dport 80 -j DNAT *

--to-destination ${my_ip}:8775

它另外的用途是虚拟机在 resize、cold migrate 等操作时，与目的端宿主机进行数据通信。该项的默认值为宿主机的外网 IP 地址，建议改为内网地址以避免潜在的安全风险。

metadata_listen = 内网地址

此项是 nova-api-metadata 服务监听的 IP 地址，可以从上面的 iptables 规则里面看出它与 my_ip 的配置项有一定的关联，保持一致是最明智的选择。

novncproxy_base_url = vncserver_proxyclient_address = ${private_ip_of_compute_host}

vncserver_listen = ${private_ip_of_compute_host}

novncproxy_host = ${private_ip_of_host}

我们仅在部分节点上部署 novncproxy 进程，并把这些进程加入到 HAProxy 服务中实现 novnc 代理进程的高可用，多个 HAProxy 进程使用 Keepalived 实施 HAProxy 的高可用，对外只需要暴露 Keepalived 管理的虚拟 IP 地址即可：

这种部署方式好处是：

1）实现 novnc 代理服务的高可用

2）不会暴露云平台相关节点的外网地址

3）易于 novnc 代理服务的扩容

但也有不足：

1）虚拟机都监听在其所在的计算节点的内网 IP 地址，一旦虚拟机与宿主机的网络隔离出现问题，会导致所有虚拟机的 VNC 地址接口暴露出去

2）在线迁移时会遇到问题，因为 VNC 监听的内网 IP 在目的端计算节点是不存在的，不过这个问题 nova 社区已经在解决了，相信很快就会合入 J 版本。

resume_guests_state_on_host_boot = true

在 nova-compute 进程启动时，启动应该处于运行状态的虚拟机，应该处于运行状态的意思是 nova 数据库中的虚拟机记录是运行状态，但在 Hypervisor 上该虚拟机没有运行，在计算节点重启时，该配置项具有很大的用处，它可以让节点上所有虚拟机都自动运行起来，节省运维人员手工处理的时间。

api_rate_limit = false

不限制 API 访问频率，打开之后 API 的并发访问数量会受到限制，可以根据云平台的访问量及 API 进程的数量和承受能力来判断是否需要打开，如果关闭该选项，则大并发情况下 API 请求处理时间会比较久。

osapi_max_limit = 5000

nova-api-os-compute api 的最大返回数据长度限制，如果设置过短，会导致部分响应数据被截断。

scheduler_default_filters = RetryFilter, AvailabilityZoneFilter, RamFilter, ComputeFilter, ImagePropertiesFilter, JsonFilter, EcuFilter, CoreFilter

nova-scheduler 可用的过滤器，Retry 是用来跳过已经尝试创建但是失败的计算节点，防止重调度死循环；AvailabilityZone 是过滤那些用户指定的 AZ 的，防止用户的虚拟机创建到未指定的 AZ 里面；Ram 是过滤掉内存不足的计算节点；Core 是过滤掉 VCPU 数量不足的计算节点；Ecu 是我们自己开发的过滤器，配合我们的 CPU QoS 功能开发的，用来过滤掉 ecu 数量不足的计算节点；ImageProperties 是过滤掉不符合镜像要求的计算节点，比如 QEMU 虚拟机所用的镜像不能在 LXC 计算节点上使用；Json 是匹配自定义的节点选择规则，比如不可以创建到某些 AZ，要与那些虚拟机创建到相同 AZ 等。其他还有一些过滤器可以根据需求进行选择。

running_deleted_instance_action = reap

nova-compute 定时任务发现在数据库中已经删除，但计算节点的 Hypervisor 中还存在的虚拟机（也即野虚拟机审计操作方式）后的处理动作，建议是选择 log 或者 reap。log 方式需要运维人员根据日志记录找到那些野虚拟机并手工执行后续的动作，这种方式比较保险，防止由于 nova 服务出现未知异常或者 bug 时导致用户虚拟机被清理掉等问题，而 reap 方式则可以节省运维人员的人工介入时间。

until_refresh = 5

用户配额与 instances 表中实际使用量的同步阈值，也即用户的配额被修改多少次后强制同步一次使用量到配额量记录

max_age = 86400

用户配额与实际使用量的同步时间间隔，也即距上次配额记录更新多少秒后，再次更新时会自动与实际使用量同步。

众所周知，开源的 nova 项目目前仍然有很多配额方面的 bug 没有解决，上面两个配置项可以在很大程度上解决用户配额使用情况与实际使用量不匹配的问题，但也会带来一定的数据库性能开销，需要根据实际部署情况进行合理设置。

计算节点资源预留

vcpu_pin_set = 4-$

虚拟机 vCPU 的绑定范围，可以防止虚拟机争抢宿主机进程的 CPU 资源，建议值是预留前几个物理 CPU，把后面的所有 CPU 分配给虚拟机使用，可以配合 cgroup 或者内核启动参数来实现宿主机进程不占用虚拟机使用的那些 CPU 资源。

cpu_allocation_ratio = 4.0

物理 CPU 超售比例，默认是 16 倍，超线程也算作一个物理 CPU，需要根据具体负载和物理 CPU 能力进行综合判断后确定具体的配置。

ram_allocation_ratio = 1.0

内存分配超售比例，默认是 1.5 倍，生产环境不建议开启超售。

reserved_host_memory_mb = 4096

内存预留量，这部分内存不能被虚拟机使用

reserved_host_disk_mb = 10240

磁盘预留空间，这部分空间不能被虚拟机使用

service_down_time = 120

服务下线时间阈值，如果一个节点上的 nova 服务超过这个时间没有上报心跳到数据库，api 服务会认为该服务已经下线，如果配置过短或过长，都会导致误判。

rpc_response_timeout = 300

RPC 调用超时时间，由于 Python 的单进程不能真正的并发，所以 RPC 请求可能不能及时响应，尤其是目标节点在执行耗时较长的定时任务时，所以需要综合考虑超时时间和等待容忍时间。

multi_host = True

是否开启 nova-network 的多节点模式，如果需要多节点部署，则该项需要设置为 True。

Keystone

配置项较少，主要是要权衡配置什么样的后端驱动，来存储 token，一般是 SQL 数据库，也可以是 memcache。sql 可以持久化存储，而 memcache 则速度更快，尤其是当用户要更新密码的时候，需要删除所有过期的 token，这种情况下 SQL 的速度与 memcache 相差很大很大。

glance

包括两个部分，glance-api 和 glance-registry，：

workers = 2

glance-api 处理请求的子进程数量，如果配置成 0，则只有一个主进程，相应的配置成 2，则有一个主进程加 2 个子进程来并发处理请求。建议根据进程所在的物理节点计算能力和云平台请求量来综合确定。

api_limit_max = 1000

与 nova 中的配置 osapi_max_limit 意义相同

limit_param_default = 1000

一个响应中最大返回项数，可以在请求参数中指定，默认是 25，如果设置过短，可能导致响应数据被截断。

OpenStack 底层依赖软件版本、配置以及性能调优

虚拟化技术选型

在私有云平台的体系架构中， OpenStack 依赖一些底层软件，如虚拟化软件，虚拟化管理软件和 Linux 内核。这些软件的稳定性以及性能关系着整个云平台的稳定性和性能。因此，这些软件的版本选择和配置调优也是网易私有云开发中的一个重要因素。

在网易私有云平台中，我们选用的是 Linux 内核兼容最好的 KVM 虚拟化技术。相对于 Xen 虚拟化技术，KVM 虚拟化技术与 Linux 内核联系更为紧密，更容易维护。选择 KVM 虚拟化技术后，虚拟化管理驱动采用了 OpenStack 社区为 KVM 配置的计算驱动 libvirt，这也是一套使用非常广泛，社区活跃度很高的一套开源虚拟化管理软件，支持 KVM 在内的各种虚拟化管理。

另一方面，网易采用开源的 Debian 作为自己的宿主机内核，源使用的是 Debian 的 wheezy 稳定分支，KVM 和 libvirt 采用的也是 Debian 社区 wheezy 源里面的包版本：

qemu-kvm 1.1.2+dfsg-6+deb7u3

libvirt-bin 0.9.12

内核选型

在内核的选型方面，我们主要考虑如下两方面的因素：

- 稳定性：在开发私有云平台的一开始，稳定性就是网易私有云开发的一大基本原则。我们采用 Debian Linux 版本，相对来说，Debian 的原生内核无疑更为稳定。这也是我们最开始的一个选择。

- 功能需求：在网易的定制开发中，为了保证虚拟机的服务性能，我们开发了 CPU QoS 技术和磁盘 QoS，它依赖底层的 CPU 和 blkio cgroup 支持。因此，我们需要打开内核中的 cgroup 配置选项。另一方面，网易私有云综合各方面考虑，将支持 LXC 这种容器级别的虚拟化，除了 cgroup 外，LXC 还依赖 Linux 内核中的 namespace 特性。

综合上述因素的考虑，我们选择了 Debian 社区的 Linux 3.10.40 内核源代码，并且打开了 CPU/mem/blkio 等 cgroup 配置选项以及 user

namespace 等 namespace 选项，自己编译了一个适配网易私有云的 Linux 内核。从使用情况来看，选择上述版本的 OpenStack 底层依赖软件后，网易私有云运行还比较稳定，我们后续还会适时的对这些软件进行更新。

配置优化

在网易私有云的稳定性得到了保障之后，我们开始了性能方面的调优工作。这一方面，我们参考了 IBM 公司的一些优秀实践，在 CPU、内存、I/O 等方面做了一些配置方面的优化。整体而言，网易私有云在注重稳定性的基础上，也会积极借鉴业界优秀实践来优化私有云平台的整体性能。

CPU 配置优化

为了保障云主机的计算能力，网易私有云开发了 CPU QoS 技术，具体来说就是采用 cfs 的时间片均匀调度，外加 process pinning 的绑定技术。

参考 IBM 的分析，我们了解到了 process pinning 技术的优缺点，并且经过测试也验证了不同绑定方式的云主机间的性能存在较大的差异。比如，2 个 VCPU 分别绑定到不同 numa 节点的非超线程核上和分配到一对相邻的超线程核上的性能相差有 30%~40%(通过 SPEC CPU2006 工具测试)。另一方面，CPU0 由于处理中断请求，本身负荷就较重，不宜再用于云主机。因此，综合上面的因素考虑以及多轮的测试验证，我们最终决定将 0-3 号 CPU 预留出来，然后让云主机在剩余的 CPU 资源中由宿主机内核去调度。最终的 CPU 配置如下所示（libvirt xml 配置）：

```
<vcpu placement='static' cpuset='4-23'>1</vcpu>
<cputune>
    <shares>1024</shares>
    <period>100000</period>
    <quota>57499</quota>
</cputune>
```

内存配置优化

内存配置方面，网易私有云的实践是关闭 KVM 内存共享，打开透明大页：

```
echo 0 > /sys/kernel/mm/ksm/pages_shared
    echo 0 > /sys/kernel/mm/ksm/pages_sharing
    echo always > /sys/kernel/mm/transparent_hugepage/enabled
    echo never > /sys/kernel/mm/transparent_hugepage/defrag
    echo 0 > /sys/kernel/mm/transparent_hugepage/khugepaged/defrag
```

经过 SPEC CPU2006 测试，这些配置对云主机 CPU 性能大概有 7%左右的提升。

I/O 配置优化

1）磁盘 I/O 的配置优化主要包含如下方面：

KVM 的 disk cache 方式：借鉴 IBM 的分析，网易私有云采用 none 这种 cache 方式。

disk io scheduler：目前网易私有云的宿主机磁盘调度策略选用的是 cfq。在实际使用过程中，我们发现 cfq 的调度策略，对那些地低配置磁盘很容易出现 I/O 调度队列过长，utils 100% 的问题。后续网易私有云也会借鉴 IBM 的实践，对 cfq 进行参数调优，以及测试 deadline 调度策略。

磁盘 I/O QoS：面对日渐突出的磁盘 I/O 资源紧缺问题，网易私有云开发了磁盘 I/O QoS，主要是基于 blkio cgroup 来设置其 throttle 参数来实现。由于 libvirt-0.9.12 版本是在 QEMU 中限制磁盘 I/O，并且存在波动问题，所以我们的实现是通过 Nova 执行命令方式写入到 cgroup 中。同时我们也开发并向 libvirt 社区提交了 blkiotune 的 throttle 接口设置 patch（已在 libvirt-1.2.2 版本中合入）来彻底解决这个问题。

2）网络 I/O 的配置优化

我们主要是开启了 vhost_net 模式，来减少网络延时和增加吞吐量。

运维经验

使用经验

- 开 源软件 bug 在所难免，但是新版本比旧版本会好用很多，尤其是对于 OpenStack 这种正在迅速成长壮大的开源软件来说更是如此，这一点在我们使用过 Essex、Folsom 和 Havana 版本后深有体会，所以建议各种 OpenStack 用户能及时的跟进社区版本，与社区保持同步。

- 不要轻易的对社区版本进行各类所谓的功能性能方面的"优化"，尤其是 在没有与社区专家交换意见之前，千万不要轻易下手，否则此类"优化"极有可能演变成故障点或者性能瓶颈点，最终可能导致无法与社区同步，毕竟一个公司或团 队（尤其是小公司、小团队）的能力和知识储备，是很难与社区成百上千的各类专家相提并论的。

- 多参考各类大型公司分享的部署架构方案，尽量不要自己闭门造车，尤其是对于开源软件来说，各类公司、团队的使用场景千差万别，各种周边组件也是应有尽有，多参考业界实践是最好的方式。

- 一些细节实现可能有很多途径，但每种方式都有优缺点，需要经过充分的论证、分析、测试验证后，才能考虑部署到生产环境使用。

- 所有的部署方案、功能设计都要考虑到平滑升级问题，即使你得到的信息是升级时可以停服，仍然要尽量避免这种情况，因为停服的影响范围很难界定。

运维准则

OpenStack 也是一个后端系统服务，所有系统运维相关的基本准则都适用，这里简单的提几点实际运维过程中根据遇到的问题总结的一些经验：

- 配置项默认值与实际环境不匹配可能导致各种问题，尤其是网络相关配置与硬件有很强的关联性，生产环境和开发环境硬件异构，导致部分默认值在生产环境不适用。应对准则：每个版本都必须在与线上硬件相同的环境测试过才能上线。

- 做好容量规划，已分配的配额量要小于云平台总容量，否则会出现各种问题，导致运维开发耗费很多不必要的精力去定位分析问题。

- 配置项过多容易出错，需要与开发人员一起仔细核对，上线时首先要通过 puppet 的 noop 功能验证改动是否正确后，才能真正上线。
- 网络规划要提前做好，如固定 IP、浮动 IP、VLAN 数量等，网络扩容难度和风险都比较大，所以提前规划好是最保险的，一个原则是大比小好，多比少好。
- 网络隔离要做好，否则用户网络安全没办法保证。
- 信息安全问题要重视，这个是老生常谈的问题了，每个平台都有相同问题，但还是要重视再重视，一旦出现安全漏洞，所有虚拟机都面临严重威胁。

原文链接：http://www.ibm.com/developerworks/cn/cloud/library/1408_zhangxl_openstack/

.NET应用架构设计—现代企业级应用分层架构核心设计要素

作者：王清培

1.背景介绍

接触分层架构有段时间了，从刚开始的朦朦胧胧的理解到现在有一定深度的研究后，觉得有必要将自己的研究成果分享出来给大家，互相学习，也是对自己的一个总结。

我们每天面对的项目结构可以说几乎都是分层结构的，或者是基于传统三层架构演 变过来的类似的分层结构，少不了业务层、数据层，这两个层是比较重要的设计点，看似这两个层是互相独立的，但是这两个层如何设计真的还有很多比较微妙的地 方，本文将分享给大家我在工作中包括自己的研究中得出的比较可行的设计方法。

2.简要回顾下传统三层架构

其实这一节我本来不打算加的，关于传统三层架构我想大家都应该了解或者很熟悉了，但是为了使得本文的完整性，我还是简单的过一下三层架构，因为我觉得它可以使得我后面的介绍有连贯性。

传统三层架构指将一个系统按照不同的职责划分层三个基本的层来分离关注点，将一个复杂的问题分解成三个互相协作的单元来共同的完成一个大任务。

1.显示层：用来显示数据或从UI上获取数据；该层主要是用来处理数据显示和特效用的，不包括任何业务逻辑。

2.业务层：业务层包含了系统中所有的核心业务逻辑，不包括任何跟数据显示、数据存取相关的代码逻辑。

3.数据层：用来提供对具体的数据源引擎的访问，主要用来直接存取数据，不包括业务逻辑处理。

其实用文字描述这三个层的基本职责还很是比较容易的，但是不同的人如何理解并 设计这三个层就形态各异了，反正我是看过很多各种各样的分层结构，各有各的特点，从某个角度讲都很不错，但是都显得有点乱，因为没有一个统一的架构模式来 支撑，代码中充满了对分层架构的理解错位的地方，比如：经常看见将“事物脚本”模式和“表模块”模式混搭使用的，导致我最后都不知道把代码写在哪里，提取 出来的代码也不知道该放到哪个对象里。

层虽简单但是要想运用的好不容易，毕竟我们还是站在一个比较高的层面比较笼统 的层面来谈论分层结构的，一旦落实到代码上就完全不一样了，用不用接口来隔离各层，接口放在哪个层里，这些都是很微妙的，当然本文不是为了说明我所介绍的 设计是多么的好，而是给大家一个可以参考的例子而已。

言归正传，三个层之间的调用要严格按照“上层只能调用直接下层，不能够越权， 而下层也不能够调用自己的上层”，这是基本的层结构的调用约束，只有这样才能保证一个好的代码结构。显示层只能调用业务层，业务层也只能调用数据层，其实 就是这么简单，当然具体的代码设计也可以大概归纳为两种，第一种是实例类或静态类直接调用；第二种是每个层之间加上接口来隔离每个层，使得测试、部署容易 点，但是如果用的不好的话效果不大反而会增加复杂度，还不如直接使用静态类来的直接点，但是用静态类来设计业务类会使多线程操作很难实施，稍微不注意就会 串值或报错。

3.企业级应用分层架构（现代分层架构的基本演变过程）

上节中我们基本了解了传统三层架构的类型和职责，本节我们来简单介绍一下现代企业应用分层架构的类型和职责。

随着企业应用的复杂度增加，在原有三层架构上逐渐演化出现在的面向企业级的分层架构，这种架构能很好的支持新的技术和代码上的最佳实践。

在传统的三层结构中的业务层之上多了一个应用层也可是说是服务层，该层是为了直接隔离显示层来调用业务层，因为现在的企业应用基本上都在往互联网方向发展，对业务逻辑的访问不会在是从进程内访问了，而是需要跨越网络来进行。

有了这一层之后会让原本显示层调用业务层的过程变得灵活很多，我们可以添加很 多灵活性在里面，更为重要的是显示层和业务层两个独立的层要想完全独立是必须要有一个层来辅助和协调他们之间的互动的。在最早的三层架构的书籍中其实也提 到了“服务层”来协调的作用，为什么我们很多的项目都不曾出现过，当我自己看到书上有讲解后才恍然大悟。（该部分可以参考：《企业应用架构模式》【马丁. 福勒】；第二部分，第9章“服务层”）

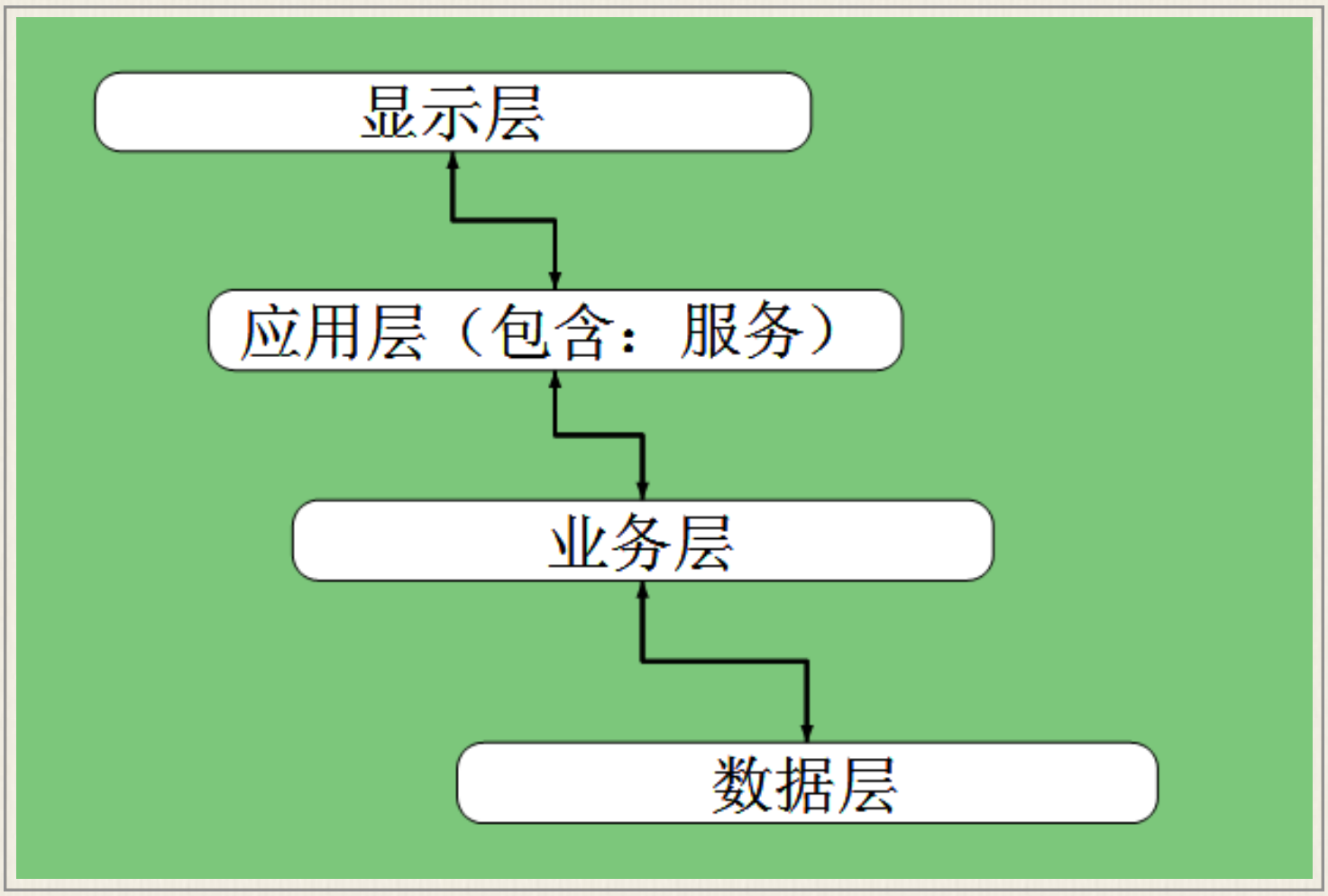


图1：（逻辑分层）

应用层中包含了服务的设计部分，应用层的概念稍微大一点，里面不仅不含了服务还包含了很多跟服务不相关的应用逻辑，比如：记录LOG，协调基础设施的接入等等，就是将服务层放宽了理解。

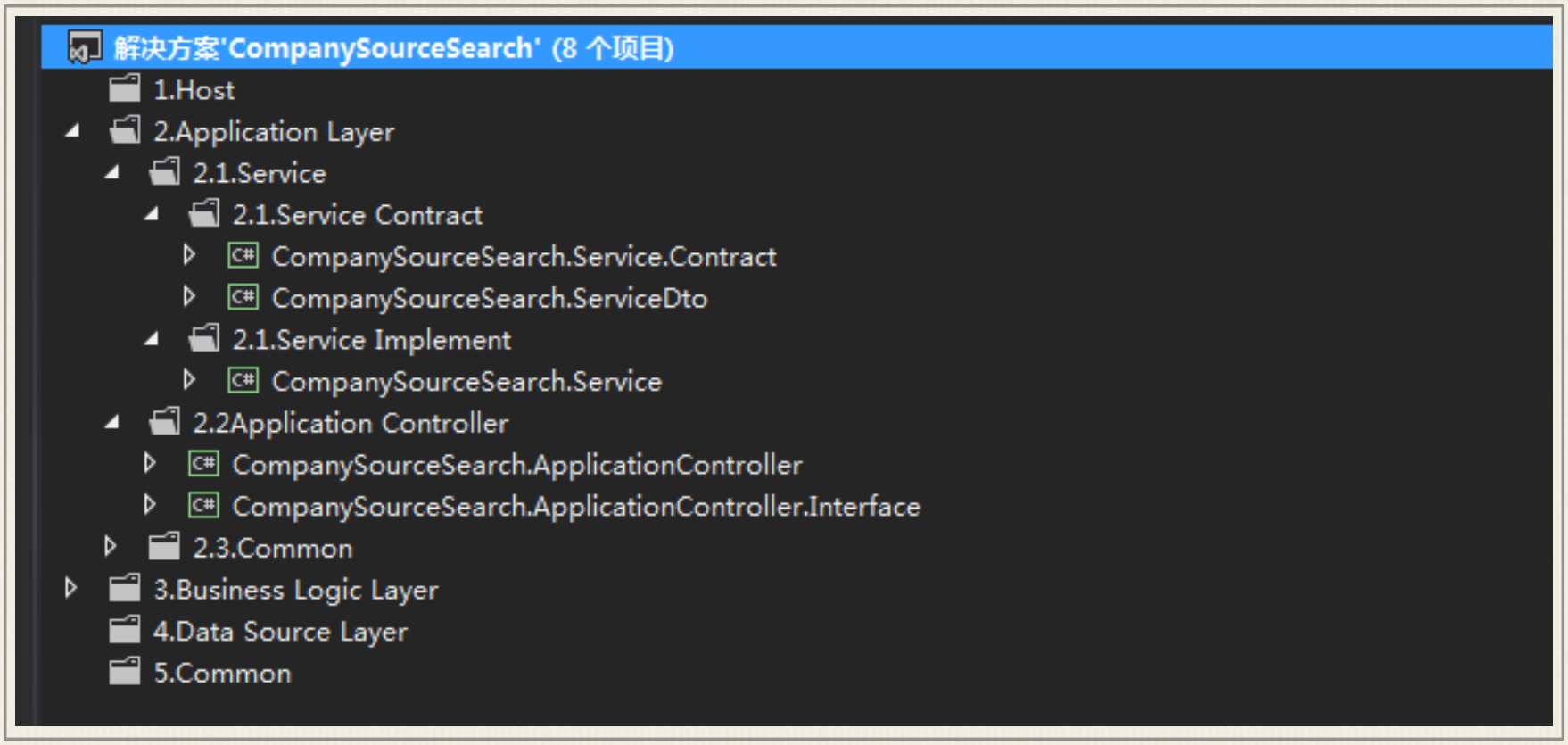


图2：（项目结构分层）

在应用层中包含了我们上述所说的”服务“，将”服务层“放宽后形成了现在分层架构中至关重要的”应用层“。应用层将负责整体的协调”业务层“和”数据层“及“基础设施”，当然还会包括系统运行时环境相关的东西。

3.1.服务层中应用契约式设计来解决动态条件不匹配错误（通过契约式设计模式来将问题在线下暴露出来）

此设计方法主要是想将动态运行时条件不匹配错误在线下自动化回归测试时就暴露出来。因为服务层中的契约可能会面临着被修改的危险性，所以我们无法知道我们本次上线的契约中是否包含了不稳定的条件不匹配的危险。

利用契约式设计模式可以在调用时自动的执行契约发布方预先设定的契约检查器，契约检查器分为前置条件检查器和后置条件检查器；我们来看一个简单的例子；

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CompanySourceSearch.Service.Contract
{
   using CompanySourceSearch.ServiceDto.Request;
  using CompanySourceSearch.ServiceDto.Response;

 public interface ISearchComputer
    {
        GetComputerByComputerIdResponse
GetComputerByComputerId(GetComputerByComputerIdRequest re-
quest);
    }
}
```

在服务契约中我定义了一个用来查询企业中电脑资源的接口，好的设计原则就是不要直接暴露查询字段而是要将其封装起来。

```
namespace CompanySourceSearch.ServiceDto
{
    public abstract class ContractCheckerBase
```

```
{
    private Func<bool> checkSpecification;
     public Func<bool> CheckSpecification
     {
         get
        {
            return this.checkSpecification;
         }
         private set
        {
            this.checkSpecification = value;
         }
 }

public void SetCheckSpecfication(Func<bool> checker)
{
    CheckSpecification = checker;
 }
 public virtual bool RunCheck()
{
   if (CheckSpecification != null)
        return CheckSpecification();

    return false;
```

```
        }
    }
}
```

然后定义了一个用来表示契约检查器的基类，这里纯粹是为了演示目的，代码稍微简单点。服务契约的请求和响应都需要通过继承这个检查器类来实现自身的检查功能。

```
namespace CompanySourceSearch.ServiceDto.Request
{
    public class GetComputerByComputerIdRequest : ContractChecker-Base
    {
        public long ComputerId { get; set; }

        public GetComputerByComputerIdRequest()
        {
            this.SetCheckSpecfication(() => ComputerId > 0/*ComputerId>0的检查规则*/);
        }
    }
}
```

Request类在构造函数中初始化了检查条件为：ComputerId必须大于0。

```
namespace CompanySourceSearch.ServiceDto.Response
{
    using CompanySourceSearch.ServiceDto;

    public class GetComputerByComputerIdResponse : ContractCheck-
erBase
    {
        public List<ComputerDto> ComputerList { get; set; }

        public GetComputerByComputerIdResponse()
        {
            this.SetCheckSpecfication(() => ComputerList != null &&
ComputerList.Count > 0);
        }
    }
}
```

同样Response类也在构造函数中初始化了条件检查器为：ComputerList不等于NULL并且Count要大于0。还是那句话例子是简单了点，但是设计思想很不错。

对前置条件检查器的执行可以放在客户端代理中执行，当然你也可以自行去执行。 后置条件检查器其实在一般情况下是不需要的，如果你能保证你所测试的数据是正确的，那么作为自动化测试是应该需要的，当时维护一

个自动化测试环境很不容 易，所以如果你用后置条件检查器来检查数据动态变化的环境时是不太合适的。

3.2.应用层中的应用控制器模式（通过控制器模式对象化应用层的职责）

应用层设计的时候大部分情况下我们都喜欢使用静态类来处理,静态类有着良好的代码简洁性,而且还能带来一定的性能提升。但是从长远来考虑静态类存在一些潜在的问题，数据不能很好的隔离，重复代码不太好提取，单元测试不太好写。

为了能够在很长的一段时间内似的项目维护性很高的情况下还是建议将应用控制器使用实例类设计，这里我喜欢使用“应用控制器”来设计。它很形象的表达了协调前端和后端的职责，但是具体不处理业务逻辑，与MVC中的控制器很像。

```
namespace CompanySourceSearch.ApplicationController.Interface
{
    using CompanySourceSearch.Service.Contract;
    using CompanySourceSearch.ServiceDto.Response;
    using CompanySourceSearch.ServiceDto.Request;

    public interface ISearchComputerApplicationController
    {
        GetComputerByComputerIdResponse
GetComputerByComputerId(GetComputerByComputerIdRequest re-
quest);
    }
}
```

在应用控制器中我们定义了一个用来负责上述查询Com-puter资源的的控制器接口。

```
namespace CompanySourceSearch.ApplicationController
{
    using CompanySourceSearch.ApplicationController.Interface;
    using CompanySourceSearch.ServiceDto.Request;
    using CompanySourceSearch.ServiceDto.Response;

    public class SearchComputerApplicationController : ISearchCom-puterApplicationController
    {
        public GetComputerByComputerIdResponse GetComputerByComputerId(GetComputerByComputerIdRequest request)
        {
            throw new NotImplementedException();
        }
    }
}
```

控制器实现类。这样可以很清晰的分开各个应用控制器，这样对服务实现来说是个很不错的提供者。

```
namespace CompanySourceSearch.ServiceImplement
{
    using CompanySourceSearch.Service.Contract;
    using CompanySourceSearch.ServiceDto.Response;
    using CompanySourceSearch.ServiceDto.Request;
    using CompanySourceSearch.ApplicationController.Interface;

    public class SearchComputer : ISearchComputer
    {
        private readonly ISearchComputerApplicationController
_searchComputerApplicationController;

        public SearchComputer(ISearchComputerApplicationController
searchComputerApplicationController)
        {
            this._searchComputerApplicationController = searchCom-
puterApplicationController;
        }

        public GetComputerByComputerIdResponse
GetComputerByComputerId(GetComputerByComputerIdRequest request)
        {
            return _ search Computer Application Controller .GetComputer-
ByComputerId (request);
        }
```

```
    }
}
```

服务在使用的时候只需要使用IOC的框架将控制器实现直接注入进来就行了，当然这里你可以加上AOP用来记录各种日志。

通过将控制器按照这样的方式进行设计可以很好的进行单元测试和重构。

3.3.业务层中的命令模式（事务脚本模式的设计模式运用，很好的隔离静态数据）

在一般的企业应用中大部分的业务层都是使用"事务脚本"模式来设计,所以这里我觉得有个很不错的模式可以借鉴一下。但是很多事务脚本模式都是使用静态类来处理的，这一点和控制器使用静态类相似了，代码比较简单，使用方便。但是依然有着几个问题，数据隔离，不便于测试重构。

将事务脚本使用命令模式进行对象化，进行数据隔离，测试重构都很方便，如果你有兴趣实施TDD将是一个不错的结构。

```
namespace CompanySourceSearch.Command.Interface
{
    using CompanySourceSearch.DomainModel;

    public interface ISearchComputerTransactionCommand
    {
        List<Computer> FilterComputerResource(List<Computer> Com-
puter);
    }
```

```
}
```

事务命令控制器接口，定义了一个过滤Computer资源的接口。你可能看见了我使用到了一个DominModel的命名空间，这里面是一些跟业务相关的且通过不断重构抽象出来的业务单元（有关业务层的内容后面会讲）。

```
namespace CompanySourceSearch.Command
{
    using CompanySourceSearch.Command.Interface;

    public class SearchComputerTransactionCommand : Command-
Base, ISearchComputerTransactionCommand
    {
        public List<DomainModel.Computer>
FilterComputerResource(List<DomainModel.Computer> Computer)
        {
            throw new NotImplementedException();
        }
    }
}
```

使用实例类进行业务代码的组装将是一个不会后悔的事情，这里我们定义了一个CommandBase类来做一些封装工作。

应用控制器同样和服务类一样使用IOC的方式使用业务命令对象。

```
namespace CompanySourceSearch.ApplicationController
{
    using CompanySourceSearch.ApplicationController.Interface;
    using CompanySourceSearch.ServiceDto.Request;
    using CompanySourceSearch.ServiceDto.Response;
    using CompanySourceSearch.Command.Interface;

    public class SearchComputerApplicationController : ISearchCom-
puterApplicationController
    {
        private readonly ISearchComputerTransactionCommand
_searchComputerTransactionCommand;
        public
SearchComputerApplicationController(ISearchComputerTransactionComm
and searchComputerTransactionCommand)
        {
            this._searchComputerTransactionCommand = searchComputer-
TransactionCommand;
        }

        public GetComputerByComputerIdResponse
GetComputerByComputerId(GetComputerByComputerIdRequest request)
        {
            throw new NotImplementedException();
        }
```

```
    }
}
```

到目前为止每个层之间坚持使用面向接口编程。

4.服务层作为SOA契约公布后DTO与业务层的Domain Model共用基本的原子类型

这里有个矛盾点需要我们平衡,当我们定义服务契约时会定义服务所使用的 DTO,而在业务层中为了很好的凝聚业务模型我们也定义了部分领域模型或者准确点讲，在事务脚本模式的架构中我们是通过不断重构出来的领域模型,它封装了 部分领域逻辑。所以当服务中的DTO与领域模型中的实体需要使用相同的原子类型怎么办？比如某个类型的状态等等。

如果纯粹的隔离两个层面，我们完全可以定义两套一模一样的原子类型来使用，但是这样会带来很多重复代码，难以维护。如果不定义两套那么又将这些共享的类型放在哪里比较合适，放在DTO中显示不合适，业务模型是不可能引用外面的东西的，如果放在领域模型中似乎也有点不妥。

这里我是采用将原子类型独立一个项目来处理的，可以类似于"CompanySourceSearch.DomainModel.ValueType"这样的一个项目，它只包含需要与DTO进行共享的原子值类型。

5.两种独立业务层职责设计方法（可以根据具体业务要求来搭配）

之前我们没有谈业务层的设计，这里我们重点讲一下业务层的设计包括与数据层的互操作。

从应用层开始考虑，当我们需要处理某个逻辑时从应用控制器开始可能就会认为直 接进入到服务层了，然后服务层再去调用数据层，其实这只是设计的一种方式而已。这样的设计方式好处就是简单明了，实现起来比较方便。但是这种方法有个问题 就是业务层始终还是依赖数据层的，业务层的变动依然会受到数据层的影响。还有一个问题就是如果这个时候你使用不是“事务脚本”模式来设计业务层的话也会自 然而然的写成过程式代码，因为

你将原本用来协调的应用控制器没有做到该做的事情，它其实是用来协调业务层和数据层的，我们并不一定非要在业务层中去调用数 据层，而是可以将业务层需要的数据从控制器中获取好然后传入到业务层中去处理，这和直接在业务层中去调用数据层是差不多的，只不过是写代码的时候不能按照过程式的思路来写了。

不管我们是使用事务脚本模式还是表模块模式或者当下比较流行的领域模型模式，都可以使用这种方法进行设计。

5.1.在应用层中的应用控制器中协调数据层与业务层的互动（业务层将绝对的独立）

我们将在应用控制器中去调用数据层的方法拿到数据然后转换成领域模型进行处理。

```
namespace CompanySourceSearch.Database.Interface
{
    using CompanySourceSearch.DatasourceDto;

    public interface IComputerTableModule
    {
        List<ComputerDto> GetComputerById(long cId);
    }
}
```

我们使用"表入口“数据层模式来定义了一个用来查询Computer的方法。

```
namespace CompanySourceSearch.ApplicationController
{
    using CompanySourceSearch.ApplicationController.Interface;
    using CompanySourceSearch.ServiceDto.Request;
    using CompanySourceSearch.ServiceDto.Response;
    using CompanySourceSearch.Command.Interface;
    using CompanySourceSearch.Database.Interface;
    using CompanySourceSearch.DatasourceDto;
    using CompanySourceSearch.Application.Common;

    public class SearchComputerApplicationController : ISearchCom-
puterApplicationController
    {
        private readonly ISearchComputerTransactionCommand
_searchComputerTransactionCommand;
        private readonly IComputerTableModule _computerTableModule;
        public
SearchComputerApplicationController(ISearchComputerTransactionComm
and searchComputerTransactionCommand,
            IComputerTableModule computerTableModule)
        {
            this._searchComputerTransactionCommand = searchComputer-
TransactionCommand;
            this._computerTableModule = computerTableModule;
        }
```

```
        public GetComputerByComputerIdResponse
GetComputerByComputerId(GetComputerByComputerIdRequest request)
        {
            var result = new GetComputerByComputerIdResponse();

            var dbComputer =
this._computerTableModule.GetComputerById(request.ComputerId);//从数
据源中获取Computer集合
            var dominModel =
dbComputer.ConvertToDomainModelFromDatasourceDto();//转换成
DomainModel

            var filetedModel =
this._searchComputerTransactionCommand.FilterComputerResource(dom
inModel);//执行业务逻辑过滤

            return result;

        }
    }
}
```

控制器中不直接调用业务层的方法，而是先获取数据然后执行转换在进行业务逻辑处理。这里需要澄清的是，此时我是将读写混合在一个逻辑项目里的，所以大部分的查询没有业务逻辑处理，直接转换成服务DTO返回即可。将读写放在一个项目可以共用一套业务逻辑模型。当然仅是个人看法。

这个是业务层将是完全独立的，我们可以对其进行充分的单元测试，包括迁移和公用，甚至你可以想着领域特定框架发展。

5.2.将业务层直接依赖数据层的关系使用IOC思想改变数据层依赖业务层（业务层将绝对独立）（比较优雅）

上面那种使用业务层和数据层的方式你也许觉得有点别扭，那么就换成使用本节的方式。

以往我们都是在业务层中调用数据层的接口来获取数据的，此时我们将直接依赖数据层，我们可以借鉴IOC思想，将业务层依赖数据层进行控制反转，让数据层依赖我们业务层，业务层提供依赖注入接口，让数据层去实现，然后在业务命令对象初始化的时候在动态的注入数据层实例。

如果你已经习惯了使用事物脚本模式来开发项目，没关系，你可以使用此模式来将数据层彻底的隔离出去，你也可以试着在应用控制器中帮你分担点事物脚本的外围功能。

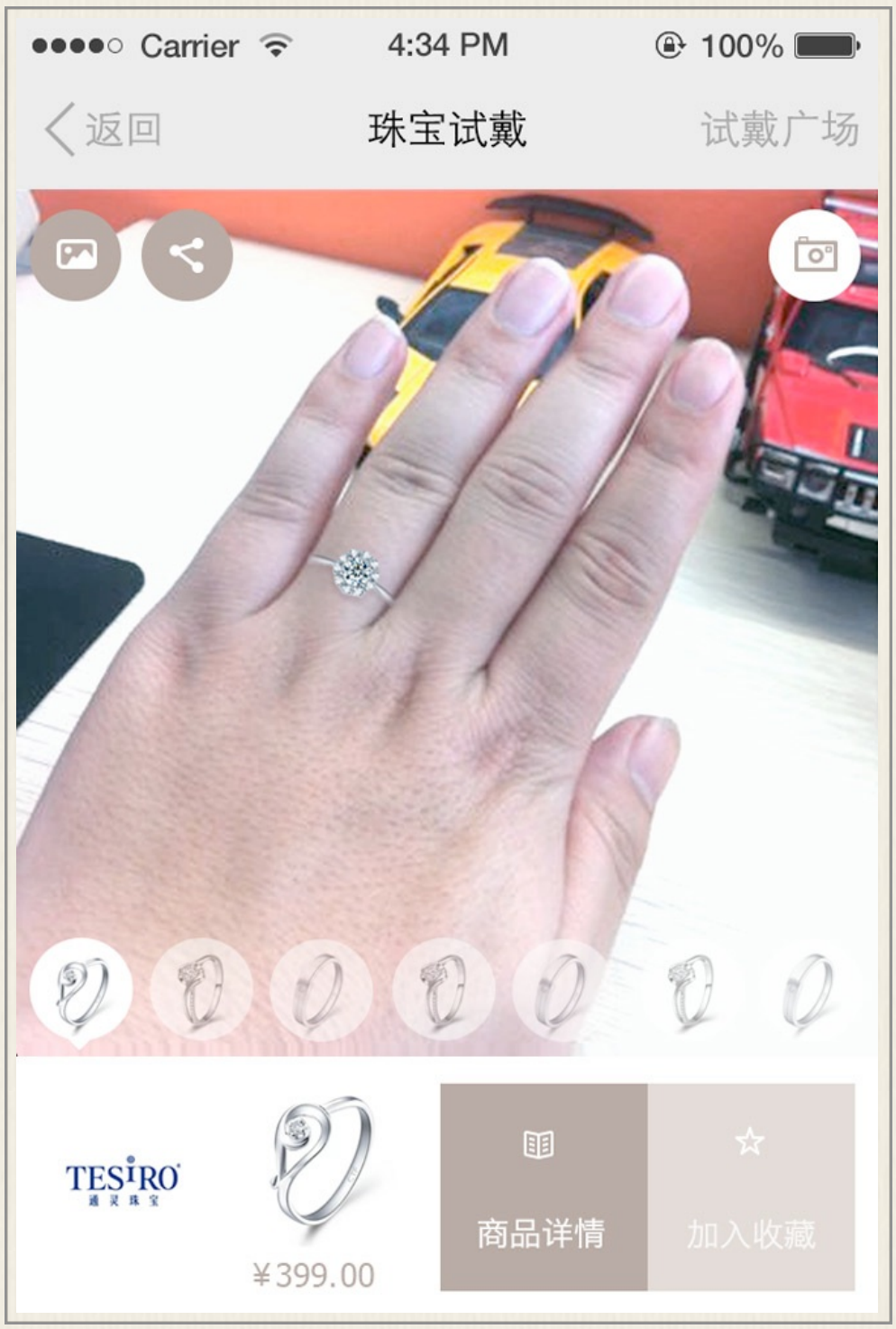
6.总结

文章中分享了本人觉得到目前来说比较可行的企业应用架构设计方法，并不能说完全符合你的口味，但是可以是一个不错的参考，由于时间关系到此结束，谢谢大家。

原文链接：http://www.cnblogs.com/wangiqngpei557/p/3923094.html?utm_source=tuicool

跨终端实践-天猫试戴的解决方案

作者：guirong



体验完产品，具体讲下技术实现方案，整体的实现过程可以分为：

拍照->获得图片数据->将商品与图片合成->生成效果图->用户保存图片

拍照

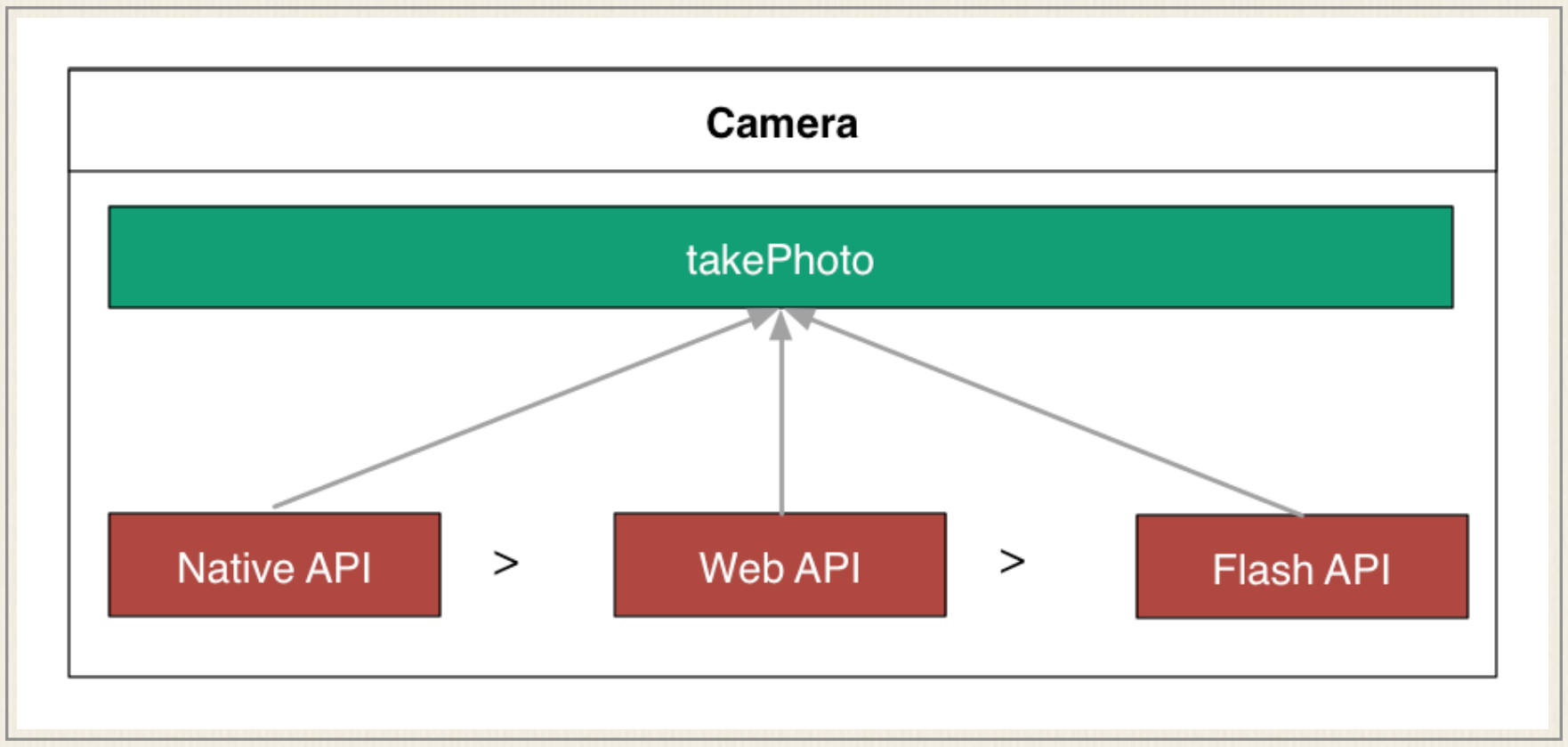
跨终端调取摄像头

这是试戴的关键一步，考虑到需要支持到各个终端，所以优先想到使用标准的web方案来实现：WebRTC-getUserMedia

基于getUserMedia，面向mobile 快速尝试，基本完成了主要的功能

但getUserMedia的支持情况并不理想，尤其是sarfari的不支持让广大ios/mac用户无法体验，这里就需要PC、mobile的兼容处理，以跨终端mobile first及优雅退化的思路设计兼容API：

1. 在浏览器中如果无法支持，将采用flash方案补齐
2. 在native中，优先采用native api 补齐



控制前后摄像头（web）

对于戒指的试戴，手机上我们期望优先调用后置摄像头，在web中启动时就需要设置优先后置摄像头，W3C文档还处于Draft阶段，相对还不是特别完善，可以通过如下设置：

```
{
    video:{
        optional: [
            {
                sourceId: $sourceId
            }
        ]
    }
}
```

通过MediaStreamTrack.getSources可获得设备的所有sourceId，注意：考虑设备可能没有外设如台式机或外设设备不可用(在虚拟机或远程)，这种情况下会报错，所以需要try&catach容错。

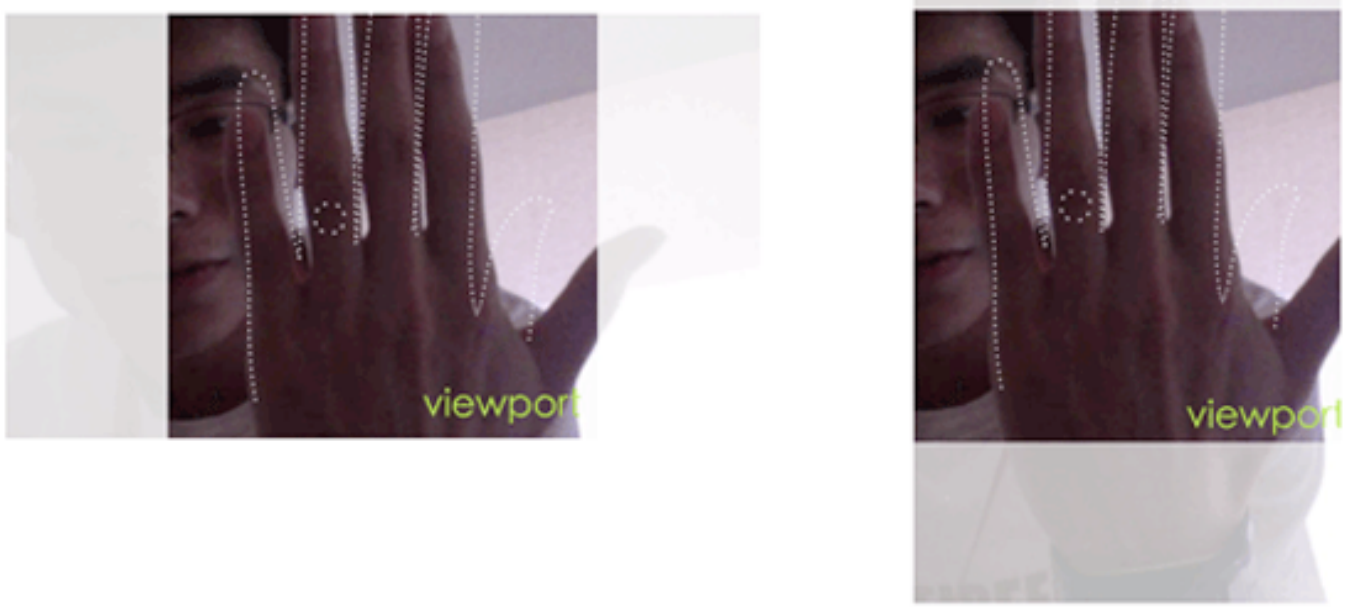
控制拍照时图片尺寸(web)

不同的终端，摄像头拍摄的图片照片尺寸是不同的，如果我们只需要获得某一部分图像，就需要对图像做剪裁，在WEB中为了不引起用户疑惑，展现给用户拍照界面时，所见最好就是所需要的部分

举个栗子：我们期望获得一个正方形的图片，但是rmbp中摄像原始是16：9的图像，考虑方案有：

1. 考虑设置video的width/height 让图像自动充满video(参考官方文档并没有规定这部分实现，最新草案也无方向，这个方案走不通)

2. 将video 部分隐藏，知道目前图像的原始尺寸，然后垂直居中，方式如图：



需要注意的是：video API中有 videoHeight及videoWidth两个属性，当video play时理论这两个属性就是当前图像的宽高，但实际情况Mozilla存在一个bug#926753，play时仍无法准确获取，兼容的方案轮训监听：

```
Event.on(video,"play",function(){
    if(this.videoHeight===0){
        return S.later(arguments.callee,100,false,this);
    }
    // now width/height ok
}
```

获取数据： 在传输大数据情况下的web与native通讯

在拍照完成native中需要把图片数据传递给web，另外用户保存图片到本地时，web又需要把合成好的图片数据传递给native让其保存，这边涉及native与web的传递大数据通讯：

1. native -> web:

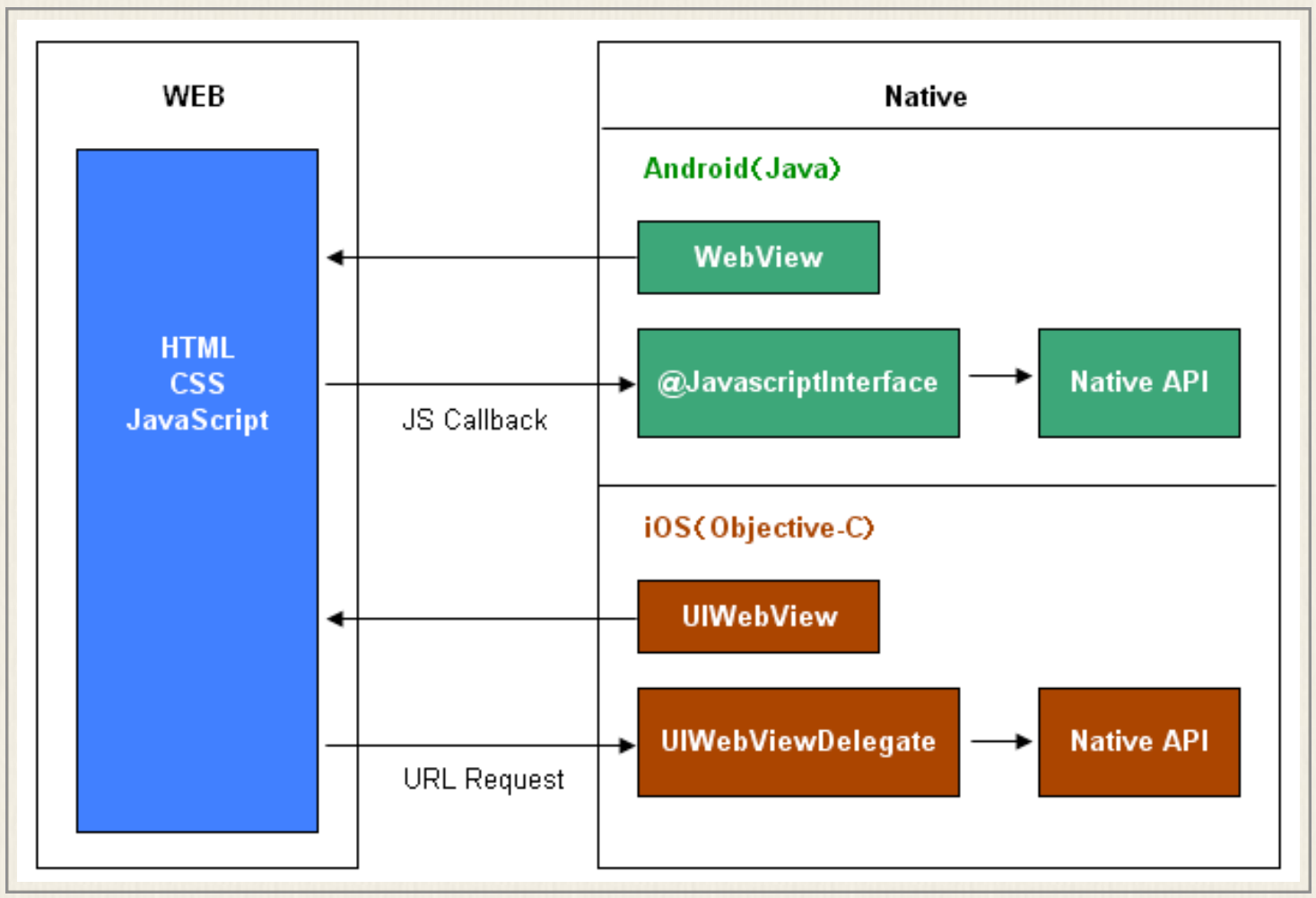
(1). android：可通过WebView#addJavascriptInterface()向web注入js方法，

(2). ios：可通过UIWebView#stringByEvaluatingJavaScriptFromString() 执行js函数，两种方式向web传递大数据都没有问题

2. web -> native:

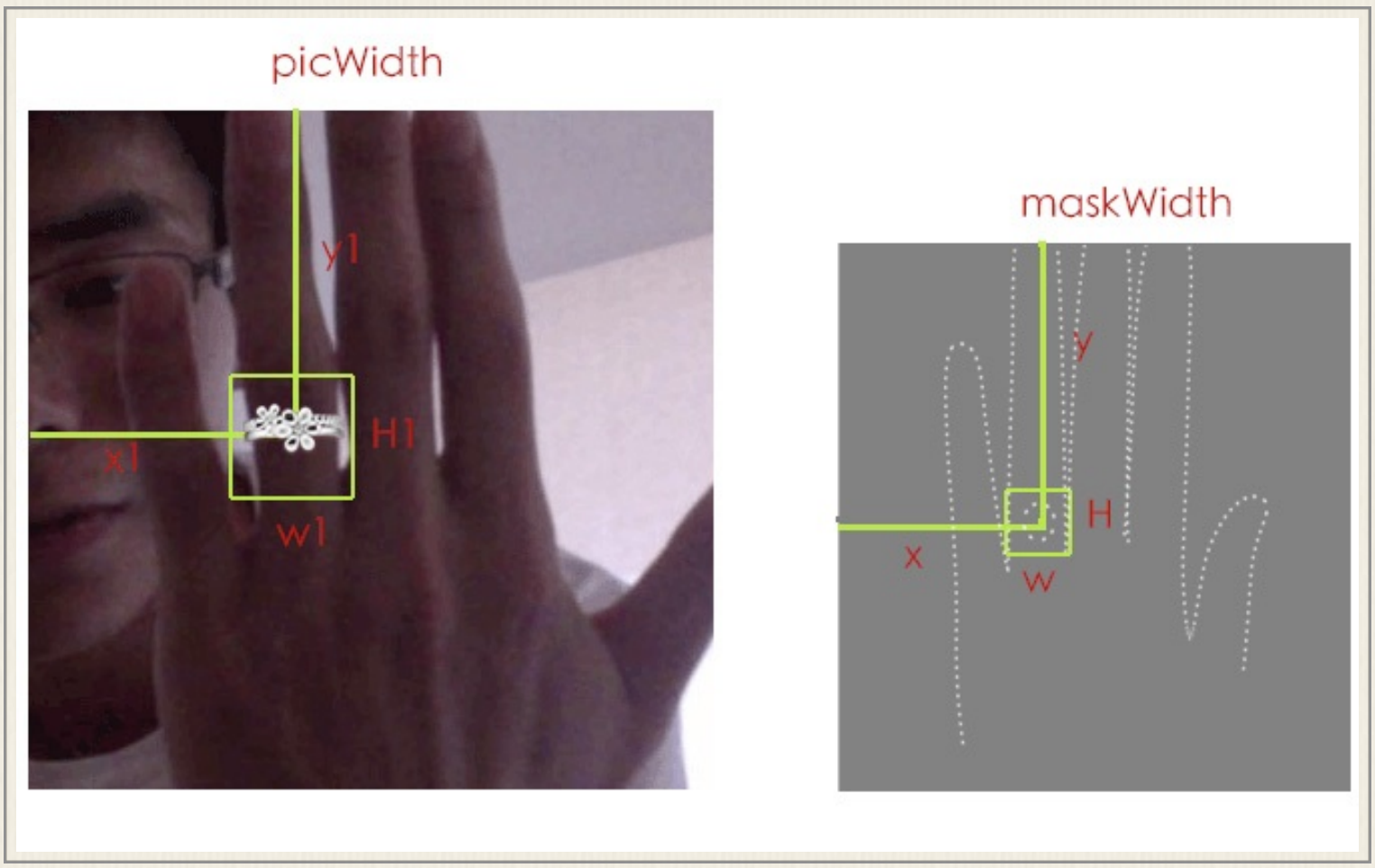
(1). android：由于可以把native类暴露给web，js可以调用暴露的方法和native通讯，大数据也没问题；

(2). ios：由于是采用UIWebViewDelegate方式，监控request方式通信，js通过自定义协议的URL用iframe请求，传输 大量数据时便存在问题；解决这个问题方案：既然native可以和web顺畅通信，就可以通知native一个js 函数名，让native自己来取。



合成展现：定位戒指位置

比较简单，用图片说明: picWidth/maskWidth = x1/x = w1/w



生成图片：Canvas图片跨域

知道了具体位置，生成图片便可以简单的调用 canvas.toDataURL获得图片数据，但这里涉及一个图片跨域问题：

Canvas获取图片数据会有跨域的限制，之前有：imageProxy flash来做代理的方案，但是这个方案仍然不够高效和简洁，尤其是对于mobile更无能为力

最好的方案是web标准的CORS，通过让服务器返回allow-origin的header，让canvas可以正常处理：

*// http response header: Access-Control-Allow-Origin: **

```
img.crossOrigin = "Anonymous";
img.onload = function() {
    ctx.drawImage( img, 0, 0 );
    canvas.toDataURL("image/png");
}
```

组件化&Mobile First

在整个开发的过程中是以组件化的思路分层处理，并封装成了具体的组件，通过封装的组件，后续拍照、试戴可以快速搭建完成，除了天猫自身业务特色的组件外，比较通用的有：

1. camera组件: 底层跨终端拍照组件，后续会移植到阿里Hybird-API一部分
2. try组件: 上层的试戴组件，处理图片位置、合并、移动旋转等，可配置不同模板及参数快速支持其他类型试戴（如眼镜）

再设计跨终端有组件时，经验是优先面向mobile设计，这样逻辑及交互流程更加简洁，可以让API涉及更加清晰，后续正对PC适当兼容。

聊业务

最后简单聊下这个业务：这是一个技术驱动业务的项目，从初期的业务重点在频道，中间经历几次业务调整，到目前把试戴作为业务后续的重点，可以说这个产品在其中起到了很多的作用，其中几点经验：

1. 不仅自己看到价值，重要让合作伙伴也看到价值
1. 在资源有限的情况下，作为技术人员去影响业务策略是不易的，认同事情的价值是关键一步
2. 快速的产出demo，验证可行性及让合作伙伴了解成本及效果
1. 有时不是业务看不到方向而是担心成本，这是技术的优势，但需要做出来看

后续

后续试戴还有很多地方可以发力，比较重要的一些方向：

1. 支持更多品类：项链、眼镜、耳环、手镯、手表等其他类目
2. 可以考虑结合图像搜索做相关推荐
3. 尝试使用图像识别，减少已有模板对比的成本，example(http://mtschirs.github.io/js-objectdetect/examples/example_sunglasses_jquery.htm)

原文链接：https://github.com/tmallfe/tmallfe.github.io/issues/4

Print —— 被埋没的Media Type

作者：solo

浪迹互联网，是否有过这样的经历，遇到自己特别需要或者特感兴趣的内容时，或收藏网址或存成笔记或将其打印出来？ 用自己喜欢的方式将内容收藏以便日后查阅。前两种应该是大家常用的处理方式，而第三者又有多少人实际操作过呢?

当你真的遇到第三者这种情况时，是否发现调出打印预览时的效果和自己的预期不太相符? 或者是网页的原样但却包含了很多在纸上不需要的内容。 这个时候你会怎么处理呢？ 放弃打印，低碳环保?复制内容到word中重新处理，费时费力费纸费电? 实际上这种情况可以很容易的被避免，那就是对网页应用–打印样式

一、原因分析

不过你可能会发现，大部分的网页都没有针对打印内容做针对处理，究其原因，大致可总结出几点

1.打印成本

首先打印机并不像电脑一样在家庭中普及， 其次纸张与墨水都属于耗材，尤其是墨水的价格直接影响着打印的成本

2.提倡无纸化

在生存环境日益恶化的今天，低碳环保是一个需要大家关注的问题， 而打印确是费纸费电的一种行为，而且与提倡的无纸化办公相违背

3.特定性较强

网页打印并不是用户的刚性需求，在一些特定的场景网页打印的机会才会多一些

二、使用场景

在网络上有哪些地方需要网页打印呢？ 可能大家会有一些印象的，比如优惠券，报销单据，简历文章，教程手册等

不过大家可以自己思考下是否还有哪些场景是需要打印的?

其实根据不同情况进行分析，需要在纸上浏览的内容，在纸上浏览可能更方便的内容，而且在纸上阅读的阅读体验及便捷程度可能比在电子设备上好， 这些都是网页打印存在的支撑。如CDC博客文章页，华尔兹规范，Prowork排期，百科词条，TAPD上的一些文档等等，还有很多需要大家自己去结合自己的业务需求来确定是否需要改进打印的体验

三、打印样式

样式引入

1.嵌入打印样式到样式文件中

```
@media print { … }
```

通用处理

1. 全局重置

    1. 优化背景色，颜色

```
* { color: #000 !important; text-shadow: none !important; ba
ckground: transparent !important; box-shadow: none !important
; }
```

快速有效的减少打印成本，全局处理后再处理具体细节。 其中!important 用于提高优先级，避免样式未生效

2. 隐藏显示

    1. 隐藏导航，广告等不相关内容，提高体验并节省成本

```
.print-hide{ display: none; }
```

    2. 显示折叠隐藏内容，将内容完整的展示在纸上

```
.print-show{ display:block; /* 恢复元素原有展现 */ }
```

3. 基本设置

    1. 字体设置，标题等突出内容使用无衬线字体加强显示，普通文本使用衬线字体提升阅读舒适度

标题:

```
.title{ font-family: hiragino sans gb, simhei, sans-serif; f
ont-weight: 500; font-size: 14pt;}
```

内容:

```
.text{ font-family: fangsong, simsun, serif; font-size: 12pt
;}
```

需要注意的是，这里的单位使用的是绝对长度单位pt，原因主要在于输出介质不同，输出到屏幕受分辨率大小影响，显示效果会不同.而输出到实际纸张为固定的尺寸。

> Points (pt): Points are traditionally used in print media (anything that is to be printed on paper， etc.). One point is equal to 1/72 of an inch. Points are much like pixels， in that they are fixed-size units and cannot scale in size.

更详细的内容可以浏览最后参考链接中的《CSS文字大小单位px、em、pt》

2. 内容展现

页边距:

```
@page { margin: 1.5cm; }
```

很好理解，和word中的类似，纸张上下左右的留白宽度

排版:

```
.type { width: 100%; margin: 0; float: none; line-height: 1.
5; }
```

这里就是根据具体需求而自定义了，有一些常用的设置，比如加宽内容的容器，清除浮动和重新设置间距等

打印分页:

为了阅读排版分页更合理，更适于阅读有几点可以参考

1. 避免将段落或图片分割成两部分
2. 避免将引用的内容打断
3. 避免将标题与内容尽量显示到同一页中

```
h1,h2,h3,h4,h5,h6{ page-break-after:avoid; page-break-i
nside:avoid }
img { page-break-inside:avoid; page-break-after:avoid;
}
blockquote,table,pre { page-break-inside:avoid }
ul,ol,dl { page-break-before:avoid }
```

元素打磨

1. 链接

    1. 显示文字链接，便于直观的了解到内容所对应网页的信息

```
a:link:after {content: " (" attr(href) ") "; /* 字体小一点、斜体等 */}
```

    2. 突出显示

```
a:link {font-weight: bold; text-decoration: underline; color: #06c; }
```

2. 图片

    1. 图片过宽超出纸张大小

```
img { max-width: 100% !important;}
```

    2. 防止图片被分隔在不同页

```
img {page-break-inside: avoid;}
```

    3. 节省打印成本


```
img {filter: url(inverse.svg#negative); -webkit-filter: invert(100%); filter: invert(100%); }
```

> Firefox对一些实验性质的图片滤镜需要svg文件支持，具体可参加filter

```
<svg xmlns="http://www.w3.org/2000/svg">
        <filter id="negative">
        <feColorMatrix values="-1 0 0 0 1
        0 -1 0 0 1
        0 0 -1 0 1
        0 0 0 1 0" />
        </filter>
 </svg>
```

应用滤镜样式后打印样式对比，图片截自CDC博客

3. 表格

    1. 每页都打印表头

```
thead{display:table-header-group;font-weight:bold}
```

    2. 每页都打印表尾

```
tfoot{display:table-footer-group;font-weight:bold}
```

彩色打印

@media print and (color) { … }

2.单独引用样式文件

<link rel="stylesheet" type="text/css" media="print" href="print.css" />

指定多个媒体类型

<link rel="stylesheet" type="text/css" media="screen, print" href="all.css" />

打印样式手册

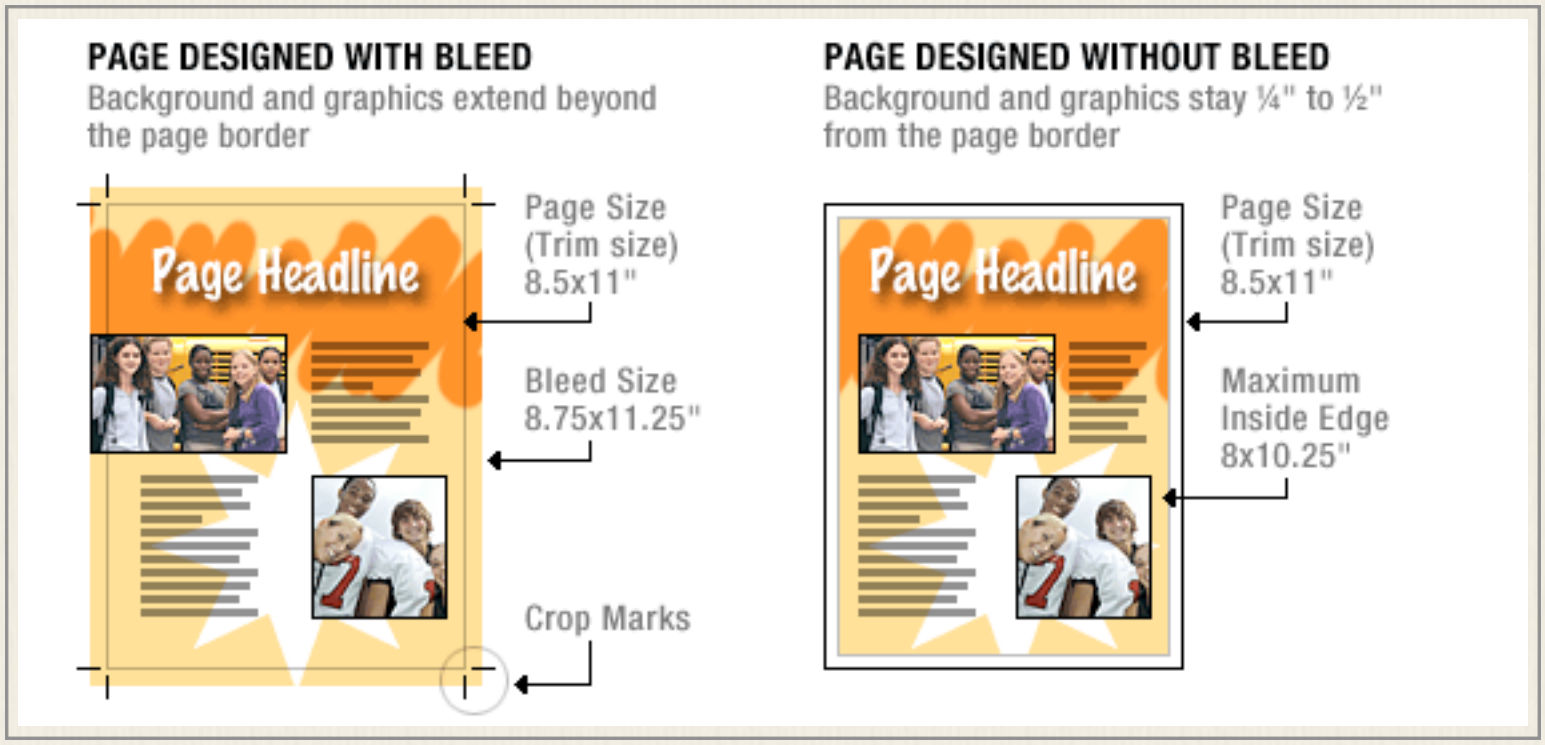
名称	描述	详细介绍
@page	设置页面容器的版式，方向，边空等	http://www.w3.org/TR/css3-page/#at-page-rule
page-break-before page-break-after page-break-inside	设置对象前后及对象本身的分页处理	http://www.w3.org/TR/2011/REC-CSS2-20110607/page.html#page-break-props

还有一些试验性质的属性用来提升打印效果，不过目前还只是一些草案，只有部分属性被浏览器支持

Widows And Orphans

传统排版印刷的细节处理，避免一些单词或者行单独显示。

Crop Marks And Page Bleed



出血是一个常用的印刷术语，印刷中的出血是指加大产品外尺寸 的图案，在裁切位加一些图案的延伸，专门给各生产工序在其工艺公差范围内使用，以避免裁切后的成品露白边或裁到内容。在制做的时候我们就分为设计尺寸和成 品尺寸，设计尺寸总是比成品尺寸大，大出来的边是要在印刷后裁切掉的，这个要印出来并裁切掉的部分就称为出血或出血位。

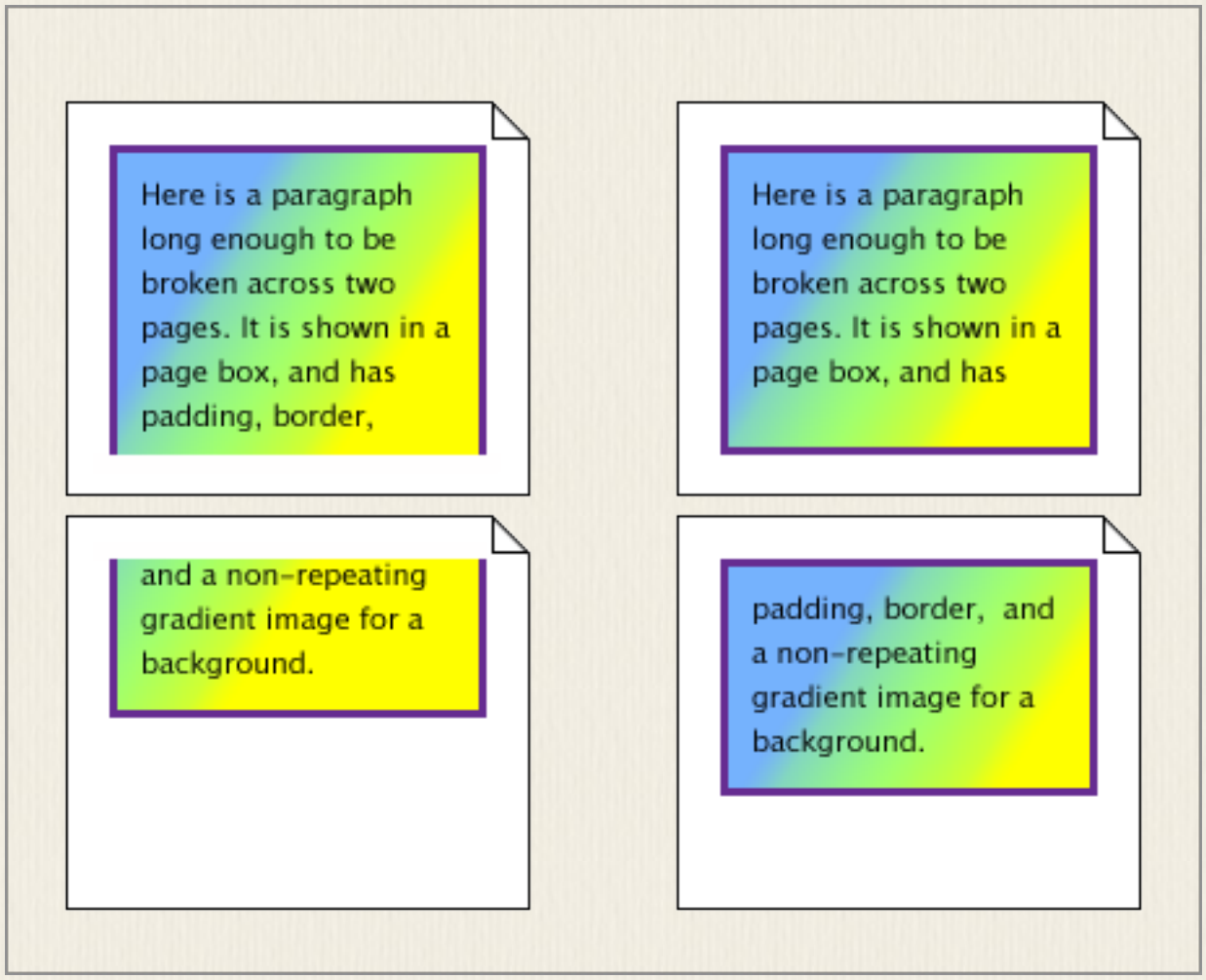
目前在一些设计图和广告海报等的打印中应用较多

Box Decoration Break

box-decoration-break: slice | clone;

左侧为slice属性，右侧为clone属性

关于打印排版的内容还有很多，感兴趣的同学还可以去w3c官网上查看下具体的工作草案CSS Paged Media Module Level 3



其它优化

- 二维码,可以便捷的访问原网址

title h3:after{ content: url(‘qrcode.png’); position: absolute; top: 0; right: 0;}

需注意的是二维码的位置不要遮盖到内容

四、打印脚本

1.客户端脚本方式打印简单高效，与打印样式配合成本最低，满足日常基本需求，或者是拼接需要打印的内容生成新页面进行打印 window.print()

2.打印控件专门针对打印进行处理，功能强大，需要安装相关插件还有就是商用需付费

五、打印实践

通过前面的简单介绍,基本可以了解了网页打印优化需要的知识,下面以CDC博客的文章详情页为例,简单介绍下优化过程

1.查看当前网页的打印效果 Ctrl+P,如下图

2.加入通用样式，根据前面提到的内容，将页头，页尾，侧边栏，评论部分还有一些赞和分享逻辑隐藏掉 display:none !important，只保留文章内容部分

3.增加内容的显示宽度，设置标题与内容字体排版 width: 100% !important

4.加入二维码提高便捷性，避免遮挡住文章内容

这样一个简单的流程就完成了打印样式的优化，实际过程非常简单。不过具体的细节部分还需要根据实际需求进一步打磨

六、优化经验

通过实践,在改进网页打印体验过程中得出了一些具体的思考

1.纸上是没有交互的

不同于网页的点击操作，纸张上的内容只适合单纯的浏览，所以一些复杂的逻辑并不适合出现在纸上。 纯粹的内容在纸上更具意义

2.节约打印成本

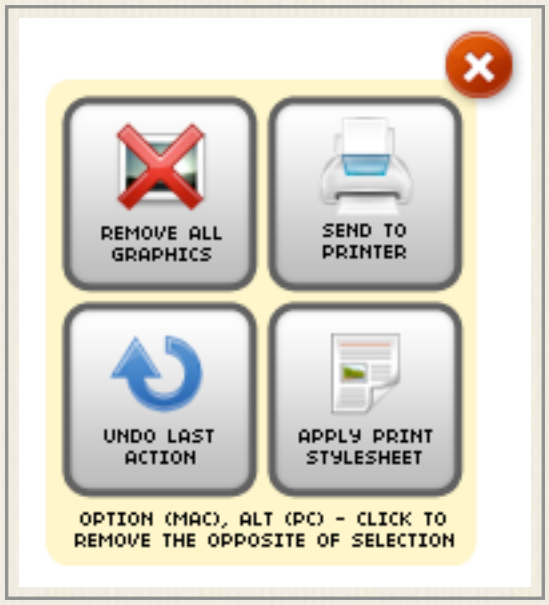
在保证内容可良好阅读的前提下尽可能的节省墨水，节约成本

3.打印提示

明确的提示和引导也是改进体验的关键，否则提供的良好体验有可能就被埋没了

七、辅助工具

1.The Print liminator



浏览器书签工具，拖至书签栏后点击，调用脚本在当前页面显示工具

- 可辅助移除页面中的图片或者不需要打印的元素
- 应用基本打印样式(比较适合英文打印)

2.Print Friendly

printfriendly
Save Money & the Environment

1. 输入网站地址，生成对应优化后的打印效果供选择打印

Save Money & the Environment

八、结语

网页打印优化的目的是为了节省成本并且可以提升基础体验，希望通过本文可以引起大家的一些关注,并通过我们的技术小力量为产品的完美体验慢慢打磨，成为精品。

原文链接：http://cdc.tencent.com/?p=8155

Swift 會不會取代 Objective-C?

作者： zonble

說到 Swift 會不會完全取代 Objective-C，目前看起來是，在 iOS 8 推出之後，應該不少應用程式會使用 Swift 開發，但是 Framework 這一層還是要用 Objective-C 來寫。

我們還是要回來看 Objective-C 與 Swift 的 run time 是怎麼實作的。無論是在 Objective-C 或 Swift 中，一個 class 有哪些 method，都是儲存在 virtual method table （以下簡稱 vtable）中，但是實作方式不一樣。

Objective-C 的 vtable 是個像是 dictionary 的實作，table 中每一筆資料都是 C 字串當 key（又稱 selector），對應到相對的 C Function pointer，所以每次呼叫某個 Objective-C 物件的某個 method，就是透過一個叫做 objc_msgSend 的 function，在表格中根據字串當 key，尋找對應該執行的 function—如果找不到，還會去問這個 Class 是否有實作 forwardInvocation: 這個 method，還是沒有的話，就會丟出找不到 method 的 exception，這部份可以參考蘋果自己的官方文件。

至於 Swift 的 vtable 則是一個 array，對某個物件呼叫某個 method 時，相當於要求執行這個 function array 當中的第幾個 function（參見 Mike Ash 的文章）。

如此就會造成以下影響：在 Swift 中，尋找要呼叫的 method 的速度，理論上就會比 Objective-C 來得快，因此同樣功能的程式，用 Swfit 寫，整體上應該會比用 Objective-C 來得快。但同時，Swift 的 vtable 中每個 function 的順序就不能改變，在編譯的時候，vatble 裡頭是什麼順序，在執行的時候就得要是什麼順序，如果是執行的時候載入了不同的 runtime，順序不對，那就會呼叫到錯誤的 function。

蘋果是怎麼確保 compile 時用到的 runtime 與執行時的 runtime 是同一份呢？就是，在編譯應用程式的時候，Xcode 會自動把一份 Swift runtime 複製到應用程式的 bundle 中；所以，你的裝置中有多少個 Swift 應用程式，就會有多少份 Swift runtime。Swift 的 runtime 本身不大，這麼做還沒什麼問題，但是你寫一個 iOS 應用程式還會用到 Foundation、UIKit…一大堆的 Framework，如果要把每個 Framework 都複製一份，聽起來就很不對，照裡說不會換成 Swift 才對，Objective-C 當初用字串當 key，某方面來看，也就是為了處理像 DLL Hell 這樣的問題。

外部開發者比較沒有機會寫到比較底層的 Framework，如果只是寫一個 iOS 應用程式的話，按照蘋果的宣稱，全都用 Swift 寫看來不是什麼問題。

而在 iOS 8 之後，蘋果也終於開放在應用程式 bundle 中，放置自己寫的、或別人包好的 Framework（Mac OS X 從一開始就開放了）。如果是自己寫的 Framework，而且這個 Framework 與主要應用程式的 Target 是在同一個 Xcode project 中，跟著主應用程式 Target 一起編譯，那應該還不會有什麼問題，但如果用的是別人已經編譯好的 Framrwork，那就相當危險—你不能夠保證別人的這個 Frame-
work，與你現在的開發環境用的是同一個版本的 Swift runtime。

iOS 8 同時也開放了一些系統 extension，這些 extension 感覺起來原理就是 Mac OS X 上的 plug-in bundle，透過 dynamic loading 讀入外部的 library。感覺起來 extension 用 Swift 寫似乎也頂危險的—一個應用程式載入一堆 plug-in，每個 plug-in 可能相依於不同的 Swift runtime，看起來就會出亂子。不過目前還沒看到蘋果自己在這方面有什麼文件。

原文链接：http://zonble.net/archives/2014_08/1624.php?utm_source=tuicool

关于 AngularJS 框架的使用有哪些经验值得分享?

作者：墨磊

最近做的某个项目的 UI 部分 Mobile Campus（Google Drive 可能需要跨墙）

代码：https://github.com/morlay/angular-mobile-ui

然后，说说我的一些做法。可能不够完善，毕竟还在折腾中。

DOM 的整体 or 零散

首先是这篇神贴：

javascript - How do I "think in AngularJS" if I have a jQuery background?(http://stackoverflow.com/questions/14994391/how-do-i-think-in-angularjs-if-i-have-a-jquery-background)

AngularJS 与 jQuery 等传统操作 DOM 的思想有所不同，

对于 jQuery 等，一般是先有完整 DOM 然后在这些 DOM 的基础上进行二次调教。

而 AngularJS 等框架则是 根据 数据模型 以及其对应的 DOM 模版，然后通过模版像搭积木那样组合页面。

显然的，前者在 SEO 上有天然优势；而后者，搜索引擎还只能拿到某个模版，而无内容。

暂时没想到有什么特别好的解决方案，或许，对于内容页，可以继续使用传统方式，而只在需要更多交互的地方应用 AngularJS，特别是在移动端应用上。

同理适用于各种 前端的 MVC 框架，后端只要为前端提供数据接口，而不再需要为其拼接 HTML.

模块化

AngularJS 也是遵循 AMD 的。（AMD 是啥，参考：使用 AMD、CommonJS 及 ES Harmony 编写模块化的 JavaScript）

虽然它也可以按照传统代码方式来写（其首页介绍的用法 AngularJS — Superheroic JavaScript MVW Framework），但是，既然都提供了这么一种模块的方法，为何不用上呢 （参考下他已有的较成熟衍生库 https://github.com/angular-ui/bootstrap）。

```
angular.module('app', [
    'moduleA',
    'moduleB',
])
.controller('MainCtrl', [
    '$scope',
    function ($scope) {

}]);
```

而且，这种写法还可以方便做代码的合并与压缩，在后面 Grunt 自动化一节中，就会提到使用 Nodejs/Grunt 来自动的做这些事情。

可复用模版 or 业务逻辑模版

今年 Google 开发者大会中 提到的 Polymer（Welcome - Polymer）

这货让人感觉像是 Angular Directives 的进化。

而 Directives 做的事，就是把一堆 DOM 封装为一条或者一组 自定义的 HTML标签，作为可复用的模版，以供组装业务调用。 Demos 可参看：Angular directives for Twitter's Bootstrap

当然，为了方便修改，很多时候在做 directive 的时候需要将 template 用 templateUrl 代替，

不用担心文件的碎片化，不利于前端加载 Grunt 自动化 一节 会提到如何合并这些碎片化的 模版。

Directives 是作为可复用的模版，

而业务逻辑则是一般是一个业务对应一个 html 及其的 controller.

作用域间的通信

上节提到了一个 html 及其的 controller，一个完整应用自然会包含很多的业务子块。

自然会有很多很多 cotrollers.

AngularJS 提供了 方法， $scope 或者 $rootScope 的 $broadcast $emit / $on

```
$scope.$emit('eventA',msg);
$scope.$broadcast('eventA',msg);

$rootScope.$on('eventA',function(event,msg){
   console.log(msg);
});
```

至于他们之间的差别，可参考这个 Demo, Chrome F12 ，你可以看到结果。

https://googledrive.com/host/0Bwdui5aYcEA9SzN2WDJ4cXZRTTA/index.html

数据池

除了作用域间的传值外，还有个方法是统一的管理一个数据池，

对于没有业务交叉的 controller，若是有公有数据的需要，都从这个数据池中取，

而这个数据池更可以直接作为和后端数据的统一交互口，及本地缓存管理的地方。

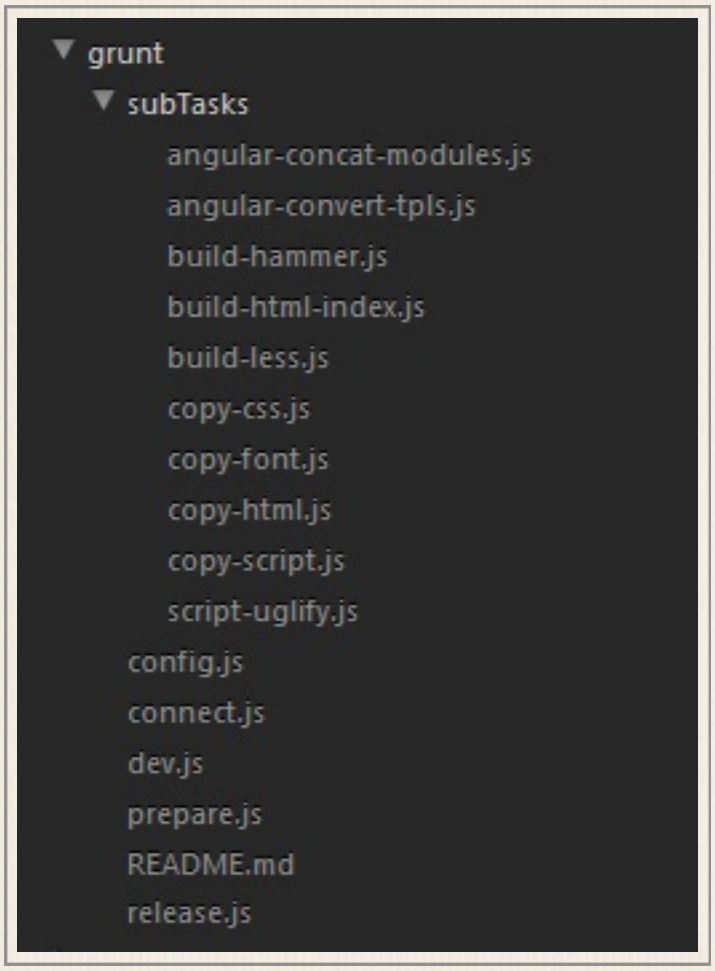
Grunt 自动化

Grunt（Grunt: The JavaScript Task Runner）出现以后，我是发现 Git 上面基本上前端相关的项目上都多了个 Gruntfile.js，可见他确实好用。

不太喜欢大多数项目中把所有任务都丢在一个文件里的方式。

所以，利用 node.js 的特性，将任务集也分解开来。

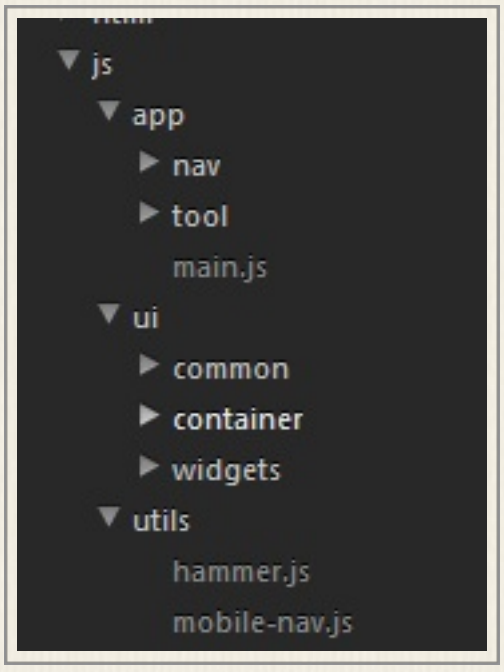
在 https://github.com/morlay/angular-mobile-ui 这里可以看到对应的代码。



这里只说说，如何按照 angular module 的依存关系自动合并对应文件。

https://github.com/morlay/angular-mobile-ui/blob/develop/grunt/subTasks/angular-concat-modules.js

首先是模块的命名，使得它能够和它的路径一致性。



```
angular.module("app.main", [
        'utils.hammer'
        , 'utils.mobile-nav'
        , 'ui.container.drawer'
        , 'ui.container.error-alert'
        , 'app.nav.nav-left'
        , 'app.nav.nav-right'
        , 'app.tool.news'
        , 'app.tool.course'
    ])
```

看这两张图就明白了。

第二，除了特殊的，全局公用的模块外，

其他模块在各自业务组件中建立引用关系。

避免载入多余的模块。

```
module.exports = function (grunt) {
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json') //
        , version: '<%= pkg.version %>' //
        , isPhoneGap: false //
        , dist: 'dist'  //
        , build: 'build'  //
        , appModule: 'app.main' //
        , modules: []  // to be filled in by build task
        , appDependencies: [
            '"ngMobile"'
        ] //
        , filename: '<%= pkg.name %>'  //
        , meta: {
            modules: 'angular.module("<%= pkg.shortName %>", ["<%= appModule %>"]);',
            all: 'angular.module("<%= pkg.shortName %>", ["<%= pkg.shortName %>-tpls","<%= appModule %>",<%= appDependencies %>]);'
        }, endInit: {
            web: 'angular.bootstrap(window.document, ["<%= pkg.shortName %>"]);',
            phoneGap: 'document.addEventListener("deviceready", function () {<%= endInit.web %>});'
        }
    }
```

比如这里，我只需要把 Grunt 配置中，

把 app.main 作为了入口文件，

并配置它的全局引用，ngMobile 和 tpls（可复用模版转换而成的 js）

通过 `grunt angular-concat-modules` 和 `grunt script-uglify`

```
1 angular.module("amu",["amu-tpls","app.main","ngMobile"]),angular.module("amu-tpls",[]).run(["$templateCache",function(a){a.put("tpls/accordion/accordion-g
2 }),b.select=function(){b.disabled||(b.active=!0)},f.addTab(b),b.$on("$destroy",function(){f.removeTab(b)}),b.active&&h(b.$parent,!0),b.$transcludeFn=d}}}
```

合并压缩自动完成。

当然，这里更是直接做了任务，`grunt release` 一条指令搞定一切。

而，对于 angular 模版转换为 js 有现成的 grunt-angular-templates 可用，

这里不细说了，详看代码。

测试工具

最后的，关于测试工具，官方有提供 Karma - Spectacular Test Runner for Javascript

但没用过，也不知道怎么用，希望有同行给予补充与介绍。

其他的，在 API 文档里写得挺详细的。

原文链接：http://www.zhihu.com/question/21497720/answer/18441053